

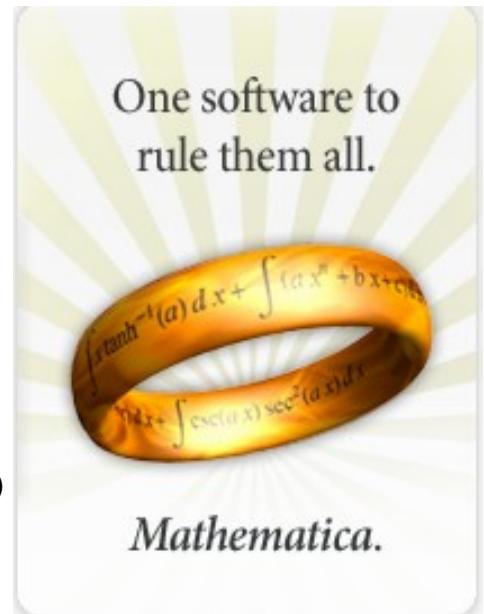
## MATEMATICA COMPUTAZIONALE (MC) 2021-2022

<https://www.unibo.it/it/didattica/insegnamenti/insegnamento/2022/479026>

[giulia.spaletta@unibo.it](mailto:giulia.spaletta@unibo.it)

[www.unibo.it/sitoweb/giulia.spaletta/AVVISI](http://www.unibo.it/sitoweb/giulia.spaletta/AVVISI)

- Per inviarmi email, usate [@studio.unibo.it](mailto:@studio.unibo.it)  
(non rispondo ad email la cui risposta è in AVVISI)
- Materiale lezioni in **VIRTUALE** (MC nel piano di studi)
- Dall'1.1.2023 UniBo ha attiva un **contratto Unlimited Site di Mathematica desktop**. Docenti e studenti possono usarlo per ricerca/didattica, installandolo sui PC dell'universita' e sui laptop personali. Si puo' scaricare la licenza dal sito seguente usando l'email istituzionale.  
<https://www.wolfram.com/siteinfo/>
- Elementary Intro to Language  
[www.wolfram.com/language/elementary-introduction/2nd-ed/](http://www.wolfram.com/language/elementary-introduction/2nd-ed/)
- Documentation Center: [reference.wolfram.com/language](http://reference.wolfram.com/language)
- Demonstration Project: [demonstrations.wolfram.com](http://demonstrations.wolfram.com)  
([demonstrations.wolfram.com/FittingACurveToFivePoints](http://demonstrations.wolfram.com/FittingACurveToFivePoints))
- Library Archive: [library.wolfram.com](http://library.wolfram.com)  
([parabolaExplorer.nb](http://parabolaExplorer.nb): [library.wolfram.com/infocenter/Courseware/7043](http://library.wolfram.com/infocenter/Courseware/7043))
- Link a Tutorial (esempio)  
Local [tutorial/SettingUpWolframLanguagePackages](http://tutorial/SettingUpWolframLanguagePackages)  
Online <https://reference.wolfram.com/language/tutorial/ModularityAndTheNamingOfThings.html#5934>



- Code Gallery: [www.wolfram.com/language/gallery/](http://www.wolfram.com/language/gallery/)
- Intro for cs students: [www.wolfram.com/language/fast-introduction-for-programmers](http://www.wolfram.com/language/fast-introduction-for-programmers)
- Intro for math students: [www.wolfram.com/language/fast-introduction-for-math-students](http://www.wolfram.com/language/fast-introduction-for-math-students)
- MathWorld (Eric Weisstein): [mathworld.wolfram.com](http://mathworld.wolfram.com)
- Wolfram|Alpha: [www.wolframalpha.com](http://www.wolframalpha.com)

SERVIZIO PER GLI STUDENTI  
CON DISABILITÀ E CON DSA



Disturbi Specifici dell' Apprendimento : dislessia, discalculia, disortografia, disgrafia

<http://www.studentidisabili.unibo.it/>

<http://www.studentidisabili.unibo.it/chi-siamo/diventare-un-tutor-all-a-pari>

BUILT-IN SYMBOL

See Also ▾

Related Guides ▾

Tutorials ▾

URL ▾

# Factorial (!)

**UPDATED**  
show changes

 $n!$ gives the factorial of  $n$ .

## ► Details

- Mathematical function, suitable for both symbolic and numerical manipulation.
- For non-negative  $n$ , the factorial is given by  $\text{Gamma}(n+1)$ .
- For integers and half-integers, Factorial automatically evaluates to exact values.
- Factorial can be evaluated to arbitrary numerical precision.
- Factorial automatically threads over lists.

## ► Background & Context

### ▼ Examples (64)

#### ► Basic Examples (7)

#### ► Scope (33)

#### ► Generalizations & Extensions (4)

#### ► Applications (6)

#### ► Properties & Relations (9)

#### ► Possible Issues (2)

#### ► Neat Examples (3)



### See Also

**Gamma** ▪ **Binomial** ▪ **Pochhammer** ▪ **Factorial2** ▪ **FactorialPower** ▪  
**Subfactorial** ▪ **QFactorial**

---

#### Tutorials

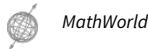
- Some Mathematical Functions
  - Combinatorial Functions
- 

#### RelatedGuides

- Combinatorial Functions
  - Discrete Mathematics
  - Recurrence and Sum Functions
  - Mathematical Functions
  - Integer Functions
  - Gamma Functions and Related Functions
  - Integer Sequences
  - Functions Used in Statistics
- 

#### RelatedLinks

- Implementation notes: Numerical and Related Functions



*MathWorld*



The Wolfram Functions Site



*An Elementary Introduction to the Wolfram Language : More about Numbers*



NKS|Online (*A New Kind of Science*)

---



## History

Introduced in 1988 (1.0) | Updated in 13 (13.0)

---

#### Cite this as

Wolfram Research (1988), Factorial, Wolfram Language function,  
<https://reference.wolfram.com/language/ref/Factorial.html> (updated 13).

## 2. Elementi fondamentali del linguaggio

### ■ Riscrittura di termini

⌘ *Mathematica* e' un sistema basato sulla riscrittura di termini (term rewriting).

Data in input una espressione (che definiremo in modo piu' preciso nel seguito), l'operazione fondamentale eseguita dal Kernel e' riconoscere quei termini che sa come sostituire con altri termini (possibilmente piu' semplici).

Per esempio, nella espressione:

```
a * a + D[a^3, a]
```

```
4 a2
```

il Kernel di *Mathematica* riscrive  $a^*a$  come  $a^2$ .

```
a^2 + D[a^3, a]
```

```
4 a2
```

Dopodiche' riscrive  $D[a^3, a]$  come  $3 a^2$ .

```
a^2 + 3 a2
```

```
4 a2
```

Infine riconosce che  $a^2+3 a^2$  puo' essere riscritto come  $4 a^2$

```
4 a2;
```

Vediamolo con `Trace[]`

```
(* // e' Postfix *)
a * a + D[a^3, a] // Trace
```

```
{ { a a, a2 }, { ∂a a3, 3 a2 }, a2 + 3 a2, 4 a2 }
```

In altre parole, il Kernel mima il modo in cui una persona esegue della matematica (il Kernel lo fa in modo completamente algoritmico).

⌘ Le **espressioni** sono l'unico tipo di oggetto in *Mathematica*: esse vengono usate per rappresentare sia il

codice che i dati.

Le espressioni hanno una struttura annidata: espressioni piu' grandi sono composte da espressioni piu' piccole, che a loro volta sono composte da espressioni via via piu' piccole, fino ad arrivare espressioni che non possono essere suddivise) del linguaggio.

Quando *Mathematica* esegue la riscrittura di termini, rimpiazza sempre una espressione con un'altra. Questa consistenza di rappresentazione e di operazione rappresenta la caratteristica piu' importante del linguaggio di programmazione in *Mathematica*.

## 2.1 Espressioni: espressioni normali (2.1.1)

⌘ Ogni cosa in *Mathematica* e' una espressione.

Esistono fondamentalmente **due** tipi di espressione: atomica e normale.

Gli **atomi** possono essere simboli, numeri o stringhe di caratteri (li vedremo meglio in 2.1.2).

Le **espressioni normali** hanno la forma:

```
Head[ part1, part2, ... ]
```

in cui Head, part1, part2, ecc. sono ciascuna una espressione.

⌘ La espressione:

```
Sin[Log[2.5, 7]];
```

e' una espressione normale:

- la sua Head e' un atomo (il simbolo Sin);
- la sua **unica** parte e' un'altra espressione normale (Log[2.5, 7]).

A sua volta:

```
Log[2.5, 7];
```

e' una espressione normale:

- la sua Head e' un atomo (il simbolo Log);
- la sue **due** parti sono il numero reale 2.5 ed il numero intero 7.

⌘ La sintassi delle espressioni e' disegnata per essere simile al costrutto di *chiamata di funzione* in guaggi quali C.

Risulta abbastanza immediato associare le Head simboliche (quali Sin o Log) a funzioni; faremo pertanto riferimento:

- alla Head di una espressione come ad una funzione;
- alle parti di una espressione come agli argomenti della chiamata alla funzione.

⌘ Non ogni espressione normale, tuttavia, puo' essere pensata come una chiamata di funzione.

Una espressione, infatti, puo' semplicemente rappresentare dati.

Ad esempio:

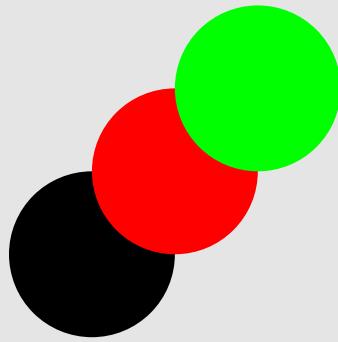
```
RGBColor[1, 0, 0]
```

e' una direttiva grafica: essa dice al Kernel che una data primitiva grafica (cui la RGBColor[1,0,0] e' associata) deve essere resa in colore rosso.

Non c'e' alcuna chiamata di funzione associata al simbolo RGBColor e l'espressione RGBColor[1,0,0] non puo' essere riscritta in alcun modo.

```
? RGBColor
```

```
Graphics[{Disk[{-1, -1}],
  RGBColor[1, 0, 0], Disk[],
  RGBColor[0, 1, 0], Disk[{1, 1}]},
 ImageSize → Small]
```



⌘ Ogni espressione in *Mathematica* puo' essere costruita usando solo **tre** blocchi di costruzione sintattica: atomi, virgolette, parentesi quadre [ ].

Riprendiamo l'esempio già visto, in cui usiamo forme speciali di input per le operazioni elementari somma e prodotto :

```
a * a + D[a^3, a];
```

Possiamo ridare lo stesso input così:

```
Plus[Power[a, 2], D[Power[a, 3], a]];
```

⌘ Il parser di *Mathematica* converte input, quali  $a^3$  in  $\text{Power}[a, 3]$  e così via.

```
(* FullForm restituisce la forma interna di una espressione *)
(* NOTA: a*a e' Power[a,2] , non e' Times[a,a] *)
a*a // FullForm
a^3 // FullForm
```

```
Power[a, 2]
```

```
Power[a, 3]
```

```
mylist = {x*y, a+a, a+b};
mylist // FullForm

(* Map applica FullForm agli elementi della lista mylist *)
Map[FullForm, mylist]
```

⌘ Forme sintattiche quali `*`, `^`, `+` sono dette *forme speciali di input* (cfr. 2.3) e servono per snellire la scrittura del codice.

Espressioni aventi (come Head) Plus, Times, Power, ecc., hanno anche *forme speciali di output*. E' per questo che l'output viene stampato in notazione matematica standard:

```
{a*a, a^3}
```

Come detto, `FullForm[]` serve ad ottenere in output la forma interna di una espressione.

```
4 a^2 // FullForm
```

## 2.1 Espressioni: atomi (2.1.2)

Un atomo, in *Mathematica*, e' una espressione che non puo' essere suddivisa in espressioni piu' piccole.

⌘ Esistono fondamentalmente tre tipi di atomi: simbolo, numero, stringa di caratteri.

### ■ Simboli

Un simbolo e' una sequenza di lettere, cifre ed il carattere \$ (tale sequenza **non** deve iniziare con una cifra).

Esempi di simboli:

```
{a,
abc,
a2,
a2b,
$a,
a$}
```

```
{a, abc, a2, a2b, $a, a$}
```

⌘ I simboli **non** sono simili alle variabili di linguaggi di programmazione, come C.

Essi sono piu' potenti, dato che non e' necessario che ad un simbolo sia stato assegnato alcun valore, al fine di poterlo usare in un calcolo.

Un simbolo *segnala* se stesso.

Un simbolo non e' meramente un sostituto (proxy) per un dato.

```
(* Esempio di calcolo simbolico.
```

```
Il risultato e' matematicamente vero, per valori arbitrari *)
```

```
a + b - 2 a
```

```
- a + b
```

Tutti i **simboli definiti da sistema** iniziano con la maiuscola o con \$

```
$MachinePrecision
```

```
$Version
```

```
15.9546
```

```
13.1.0 for Linux x86 (64-bit) (June 16, 2022)
```

```
$Version = 3
```

 **Set:** Symbol \$Version is Protected.

```
3
```

#### ■ Numeri

#### ■ Stringhe di caratteri

#### ■ Esercizio 1S pg. 25S

# Starting Out : Elementary Arithmetic

As first examples, let us look at elementary arithmetic.

## Add numbers

```
(* Plus[1234,5678] *)
1234 + 5678

(* Times[1234,5678] *)
1234 * 5678
1234 × 5678

(* Naming conventions: names cannot begin with a number *)
{a2, a 2, 2 a}

(* Information["*Solve*"] *)
? *Solve*
? Solve*
? *Solve
? Solve

(* Set[npi,N[π]] *)
npi = N[π]
FullForm[npi]
3.14159
3.141592653589793` 

FullForm[a + b]
Plus[a, b]

(* Equal[5-2,5+(-2)] *)
5 - 2 == 5 + (-2)
True

(* Equal is also used to define equations *)
a x^2 + b x + c == 0
```

## Vocabulary

$2 + 2$	<b>addition</b>	Plus
$5 - 2 == 5 + (-2)$	<b>subtraction</b>	Plus[5 ,Times[ -1 ,2 ]]

```

2*3 == 2 3      multiplication      Times
6/2 == 6 (2^-1)  division          Times[6, Power[2, -1]]
3^2  raising to a power (e.g. squaring) Power
5!              factorial         Factorial
FullForm
Trace
Sum
N
$MachinePrecision
Information (?)

```

---

## Exercises

You can check your answers in the Wolfram Cloud

**1.1** Compute  $1 + 2 + 3$  (Plus, Sum)  $\Rightarrow 6$

```

1 + 2 + 3
Plus[1, 2, 3]
Sum[i, {i, 3}]

```

**1.2** Add the numbers  $1, 2, 3, 4, 5 \Rightarrow 15$

**1.3** Multiply the numbers  $1, 2, 3, 4, 5$  (Times, Factorial)  $\Rightarrow 120$

**1.4** Compute 5 squared (i.e.  $5^2$  or 5 raised to the power 2: Power) $\Rightarrow 25$

**1.5** Compute 3 raised to the fourth power  $\Rightarrow 81$

**1.6** Compute 10 raised to the power 12  $\Rightarrow$  a trillion

**1.7** Compute 3 raised to the power  $7^8$  (Operation priorities)

**1.8** Add parentheses to  $4 - 2^3 + 4$  to make 14 (Operation priorities)

**1.9** Compute 29 thousand multiplied by 73  $\Rightarrow 2117000$

**+1.1** Add all integers from - 3 to + 3 (Sum)  $\Rightarrow 0$

**+1.2** Compute 24 divided by 3  $\Rightarrow$  8 (Division, Power, Reciprocal)

```

{24/3,
 24 (1/3),
 24 * 3^-1,
 Times[24, Power[3, -1]]}

{8, 8, 8, 8}

(* x/y is equivalent to x * y^(-1) *)
x/y == x * y^(-1)
FullForm[x/y]

True

Times[x, Power[y, -1]]

```

### +1.3 Compute 5 raised to the power 100 (Function composition)

**+1.4** Subtract 5 squared from 100  $\Rightarrow$  75 (Various ways to get 75 from 5 and 10)

**+1.5** Multiply 6 by 5 squared, and add 7  $\Rightarrow 157$  (Operation priorities, Trace)

```
6 × 52 + 7;  
(* Power has priority on Times, that has priority on Plus *)  
Trace[6 × 52 + 7]  
{ {{52, 25}, 6 × 25, 150}, 150 + 7, 157 }
```

**+1.6** Compute  $3^2 - 2^3$  (Trace)  $\Rightarrow 1$

**+1.7** Compute  $2^3 \times 3^2$  (Trace)  $\Rightarrow 72$

**+1.8** Compute "double the sum of 8 and negative 11"  $\Rightarrow$  -6

# Q & A

How to avoid getting fractions in a division (i.e. how to get a Real) ?

## What happens if I compute $1/0$ ?

1 / 0

 **Power:** Infinite expression  $\frac{1}{0}$  encountered.

## ComplexInfinity

# Introducing Functions

`2+2` is understood as `Plus[2,2]`.

`Plus` is a function.

- There are more than 5000 **built-in** functions in *Mathematica*.

Arithmetic uses very few of these.

**Compute  $3 + 4$  using `Plus`:**

```
Plus[3, 4]
```

**Compute  $1 + 2 + 3$  using `Plus`:**

```
Plus[1, 2, 4]
```

**Times** does multiplication :

```
Times[2, 3]
```

You can put functions inside other functions:

```
Times[2, Plus[2, 3]]
```

- All functions use **square** brackets for their arguments.
- A function name starts with a **Capital** letter.

If the name is compound, each word starts with a Capital letter.

```
? LinearSolve
```

**Max** finds the maximum, or largest, of a collection of numbers.

```
Max[2, 7, 3]
```

**RandomInteger** picks a random integer between 0 and a specified N.

```
(* Pick a random integer between 0 and 100 *)
RandomInteger[100]
(* At each evaluation, you get another random number *)
RandomInteger[100]
```

100

39

```
(* You can specify a seed :
  use SeedRandom[3] in exercises involving Random *)
myseed = SeedRandom[3];
RandomInteger[100]
```

61

```
(* Organising inputs *)
{test = 4, Set[test, 4], 6 * 3, 6 * 3, Times[6, 3]}
{4, 4, 18, 18, 18}

test = 4; Set[test, 4]; 6 * 3
6 * 3
Times[6, 3]
18
18
18

(* Observe Naming and Color conventions *)
 nome2 = 3
2 nome
(* avoid special character :
   $name reserved for System constants *)
$MachinePrecision
```

## Vocabulary

Plus[2,2]	2+2	addition
Subtract[5,2]	5-2	subtraction
Times[2,3]	2*3, 2 3	multiplication
Divide[6,2]	6/2	division
Power[3,2]	3^2	raising to a power
Max[3,4]		maximum (largest)
Min[3,4]		minimum (smallest)
RandomInteger[10]		random integer from 1 (default) to 10
SeedRandom		
Print		
Set (=)		
TreeForm		
LinearSolve		
tutorial/BasicObjects		

## Exercises

### 2.1 Compute $7+6+5$ using Plus $\Rightarrow$ 18

**2.2** Compute  $2 \times (3+4)$  using Times and Plus  $\Rightarrow 14$

**2.3** Use Max to find the larger of  $6 \times 8$  and  $5 \times 9 \Rightarrow 48$

**2.4** Use RandomInteger to generate a random number between 0 and 1000

```
SeedRandom[3];
(* RandomInteger[List[0,1000]] *)
RandomInteger[{0, 1000}]
490
```

**2.5** Use Plus and RandomInteger to generate a number between 10 and 20 (is Plus really needed here?)

**+2.1** Compute  $5 \times 4 \times 3 \times 2$  using Times (Plus[] is 0; Times[] is 1)  $\Rightarrow 120$

**+2.2** Compute  $2 - 3$  using Subtract  $\Rightarrow -1$

**+2.3** Compute  $(8+7) \times (9+2)$  using Times and Plus  $\Rightarrow 165$

**+2.4** Compute  $(26-89)/9$  using Subtract and Divide  $\Rightarrow -7$

**+2.5** Compute  $100 - 5^2$  using Subtract and Power  $\Rightarrow 75$

**+2.6** Find the larger of  $3^5$  and  $5^3$  (Print)  $\Rightarrow 5^3$

```
Print[
  Power[3, 5],
  " is 3^5,      ",
  Power[5, 3],
  " is 5^3,      and their max is ",
  Max[3^5, 5^3]
];
243 is 3^5,      125 is 5^3,      and their max is 243
```

**+2.7** Multiply 3 and the larger of  $4^3$  and  $3^4$  (Max)  $\Rightarrow 3 \times 3^4$

**+2.8** Add two random numbers, each between 0 and 1000 (Set).

```
range = {0, 1000};
SeedRandom[3];
ri1 = RandomInteger[range];
ri2 = RandomInteger[range];
ri1 + ri2
```

530

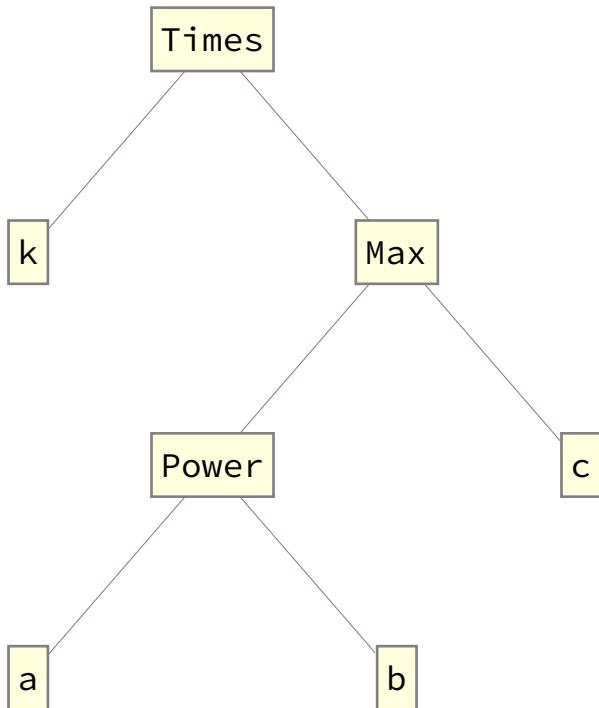
```
(* = is the special form for Set *)
Set[range, {0, 1000}];
SeedRandom[3];
Set[ri1, RandomInteger[range]];
Set[ri2, RandomInteger[range]];
Plus[ri1, ri2]
530
```

## Tech Notes

```
(* two atoms and two expressions *)
2
a
a+2
Plus[a, 2]
```

An **expression** (see § 33) consists of nested trees of functions.

```
expr = k Max[a^b, c]
TreeForm[expr]
k Max[a^b, c]
```



Plus is an n-ary operator. Subtract is a binary operator

Plus can add **any** number of numbers.

Subtract only subtracts **one** number from **another**,  
to avoid ambiguities between  $(2 - 3) - 4$  and  $2 - (3 - 4)$ .

`Plus[1, 2, 3]`

`Subtract[1, 2, 3]`

`? Subtract`

## 2.1 Representation of a Real

Fundamental result on the representation of real numbers.

**Proposition 2.1** *Every real number  $\alpha \neq 0$  can be expressed in a unique way in the following form:*

$$\alpha = \pm(a_0\beta^0 + a_1\beta^{-1} + a_2\beta^{-2} + a_3\beta^{-3} + \dots\dots\dots)\beta^p := \pm m\beta^p. \quad (2.1)$$

The integer number  $\beta > 1$  is called **base** of the representation.

The integer number  $p$  is called **exponent** of  $\alpha$ .

$\pm$  is the **sign**, positive or negative, of  $\alpha$ .

The *digits*  $a_i$  are integer numbers that satisfy the conditions:

$$0 \leq a_i < \beta \quad \forall i, \quad a_0 \neq 0, \quad (2.2)$$

$a_0$  is called *the most significant digit*.

The real number  $m > 0$  is called **mantissa** of  $\alpha$ :

$$m := a_0\beta^0 + a_1\beta^{-1} + a_2\beta^{-2} + a_3\beta^{-3} + \dots = \sum_{i=0}^{\infty} a_i\beta^{-i} \quad (2.3)$$

and it satisfies the conditions:

$$1 \leq m < \beta. \quad (2.4)$$

By exploiting the positional notation, the real  $\alpha \neq 0$  gets represented as:

- **Scientific Form**

$$\pm a_0 \bullet a_1 a_2 \dots \times \beta^p$$

The symbol (point) • is called **radix point**.

**Example 2.2** The real number  $\pi$  can be represented in base  $\beta = 10$  as follows

$$\pi = +(3 \cdot 10^0 + 1 \cdot 10^{-1} + 4 \cdot 10^{-2} + 1 \cdot 10^{-3} + 5 \cdot 10^{-4} + \dots) \cdot 10^0$$

It can be expressed in scientific form

$$\pi = +3.1415\dots \times 10^0$$

## 2.2 Floating Point Representation

In a computer, we cannot operate with real numbers, as they are represented, in general, with an unlimited number of digits. We must consider their *approximations*, i.e. numbers that are still in the form

$$\pm m \beta^p$$

but their mantissa  $m$  is represented with a **finite** number of digits.

**Example 2.4**  $\pi$  represented in base  $\beta = 10$  in scientific form:

$$\pi = +3.1415\dots \times 10^0$$

has infinite digits in its mantissa. With a finite number of digits, we may only approximate it. For example, if we have only 5 digits available (in our computer), the following happens:

$$\pi \approx +3.1415 \times 10^0, \quad \pi \neq +3.1415 \times 10^0$$

**Definition 2.1** Given an integer number  $t > 0$ , to each real number  $\alpha$ , a **finite number** gets associated

$$fl(\alpha) = \begin{cases} \pm 0 \beta^0 & \text{if } \alpha = 0, \\ \pm m_t \beta^p & \text{if } \alpha \neq 0. \end{cases} \quad (2.5)$$

$m_t$  is the **chopping** of  $m$  to the  $t$ -th digit  $a_{t-1}$  :

**Example 2.5**  $\beta = 10$ ,  $t = 3$ .

$$\alpha = 17.854 = 1.7854 \times 10^1 \longrightarrow fl(\alpha) = 1.78 \times 10^1 = 17.8$$

$$a_0 = 1, \quad a_1 = 7, \quad a_2 = 8, \quad a_3 = 5, \quad a_4 = 4$$

Formally

$$m_t := a_0\beta^0 + a_1\beta^{-1} + a_2\beta^{-2} + \dots + a_{t-1}\beta^{-(t-1)} = \sum_{i=0}^{t-1} a_i\beta^{-i} \quad (2.6)$$

$\beta$  base, for which it holds:  $1 \leq m_t < \beta$

$p$  exponent,

the integer digits  $a_i$  verify  $0 \leq a_i < \beta \quad \forall i = 0, \dots, t-1$ , with  $a_0 \neq 0$ .

The notation (2.5)–(2.6) is also called *normalized* floating point form.

### 2.2.1 Chopping Versus Rounding

If the base  $\beta$  is an even number, then the (computer) systems make use of **rounding** to the  $t$ -th digit. Rounding is obtained by performing the following operations on the mantissa  $m$  of the real number  $\alpha$  to be approximated:

- the quantity  $\beta/2$  is added to the  $(t + 1)$ -th digit  $a_t$  of  $m$
- a new number is obtained, that has mantissa

$$m + \frac{\beta}{2} \beta^{-t}$$

and that gets chopped to  $t$  digits.

**Note.** In what it follows,  $m_t$  indicates the  $t$ -digits mantissa of  $fl(\alpha)$ , independently on the fact that it has been obtained by chopping ( $fl_c$ ) or rounding ( $fl_r$ ).

Summarising, for the non-zero real number

$$\alpha = \pm a_0 . a_1 \dots a_{t-1} a_t a_{t+1} a_{t+2} a_{t+3} \dots \times \beta^p$$

we have that:

$$fl(\alpha) = \begin{cases} fl_c(\alpha) = \pm a_0 . a_1 \dots a_{t-1} \times \beta^p \\ fl_r(\alpha) = \pm fl_c\left(a_0 . a_1 \dots a_{t-1} \left(a_t + \frac{\beta}{2}\right)\right) \times \beta^p \\ \quad = \pm \hat{a}_0 . \hat{a}_1 \dots \hat{a}_{t-1} \times \beta^{\hat{p}} \end{cases} \quad (2.7)$$

As side-effect of rounding, we may have:

- a new exponent  $\hat{p}$
- different integer digit  $\hat{a}_i$ , which still verify  
 $0 \leq \hat{a}_i < \beta \quad \forall i = 0, \dots, t-1$ , with  $\hat{a}_0 \neq 0$ .

**Example 2.6**     $\beta = 10$ ,     $t = 4$ .

$$\alpha = 17.850 = 1.7850 \times 10^1, \quad fl_c(\alpha) = 17.85 = 1.785 \times 10^1 = fl_r(\alpha).$$

$$\alpha = 17.854 = 1.7854 \times 10^1, \quad fl_c(\alpha) = 17.85 = 1.785 \times 10^1 = fl_r(\alpha).$$

$$\begin{aligned} \alpha &= 17.855 = 1.7855 \times 10^1, & fl_c(\alpha) &= 17.85 = 1.785 \times 10^1, \\ && fl_r(\alpha) &= 17.86 = 1.786 \times 10^1. \end{aligned}$$

$$\begin{aligned} \alpha &= 17.859 = 1.7859 \times 10^1, & fl_c(\alpha) &= 17.85 = 1.785 \times 10^1, \\ && fl_r(\alpha) &= 17.86 = 1.786 \times 10^1. \end{aligned}$$

## 2.3 Consecutive Finite Numbers

Given the real number  $\alpha$ , let  $x$  and  $y$  be two finite numbers, both having a  $t$  digit mantissa and *consecutive* (i.e. there are no other finite numbers between  $x$  and  $\alpha$  and between  $\alpha$  and  $y$ ) such that

$$\begin{aligned}
 x &\leq \alpha < y, \\
 \alpha &= \pm a_0 . a_1 a_2 \dots a_{t-1} \quad a_t \quad a_{t+1} a_{t+2} a_{t+3} \dots \quad \times \beta^p \\
 x &= \pm a_0 . a_1 a_2 \dots a_{t-1} \quad \times \beta^p \\
 y &= \pm a_0 . a_1 a_2 \dots (a_{t-1} + 1) \quad \times \beta^p
 \end{aligned} \tag{2.8}$$

Let us consider the middle point  $M = (x + y)/2$  of the interval  $[x, y]$ . We have:

$$fl_c(\alpha) \equiv x, \quad fl_r(\alpha) \equiv \begin{cases} x & \text{if } \alpha < M \\ y & \text{if } \alpha \geq M \end{cases}$$

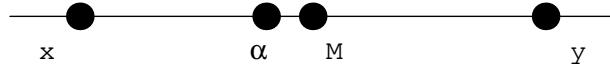


Figure 2.1: Each finite number represents itself and a whole *interval* of real numbers.

**Example 2.7**  $\beta = 10, t = 5$ .



Figure 2.2: With chopping, the finite number  $27.182 = 2.7182 \times 10^1$  represents itself and all real numbers  $\alpha$  such that  $27.182 \leq \alpha < 27.183$



Figure 2.3: With rounding, the finite number  $27.182 = 2.7182 \times 10^1$  represents itself and all reals  $\alpha$  such that  $27.1815 \leq \alpha < 27.1825$

## 2.4 Set of Finite Numbers

The symbols  $\beta$  and  $t$  denote, respectively, base and number of digits in the mantissa. The exponent  $p$  belongs to an interval, whose limiting points are integer numbers  $p_{min}$  and  $p_{max}$ :

$$p_{min} \leq p \leq p_{max}.$$

The four parameters  $\beta$ ,  $t$ ,  $p_{min}$ ,  $p_{max}$  identify the set of all finite numbers on which a computer operates:

$$F(\beta, t, p_{min}, p_{max}).$$

This finite set of numbers substitutes, in practice, the infinite real numbers.

**Example 2.8** The positive finite numbers of the set

$$F(\beta, t, p_{min}, p_{max}) \equiv F(2, 3, -2, 1)$$

are all numbers that can be represented in the form

$$a_0 \cdot a_1 a_2 \times 2^p$$

with  $a_0 = 1$ ,  $a_1$ ,  $a_2$  binary digits  
and with  $-2 \leq p \leq 1$  i.e.  $p = -2, -1, 0, 1$ .

Let us examine the mantissae, formed by the digits

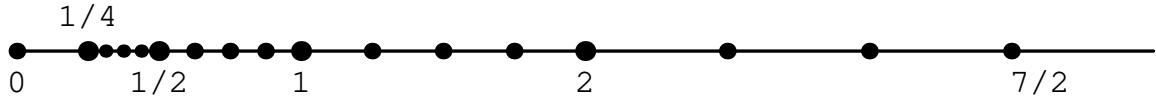
$$a_0 = 1, a_1 = 0, 1, a_2 = 0, 1 :$$

$$\begin{aligned}1 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} &= 1 \\1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} &= 1 + 1/4 = 5/4 \\1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} &= 1 + 1/2 = 3/2 \\1 \times 2^0 + 1 \times 2^{-1} + 1 \times 2^{-2} &= 1 + 1/2 + 1/4 = 7/4\end{aligned}$$

For each exponent, we have 4 mantissae (i.e. we have 4 finite numbers).  
For example:

$$\begin{aligned}1.00 \times 2^{-2} &= 1 1/4 = 1/4 = 4/16 \\1.01 \times 2^{-2} &= (5/4) (1/4) = 5/16 \\1.10 \times 2^{-2} &= (3/2) (1/4) = 3/8 = 6/16 \\1.11 \times 2^{-2} &= (7/4) (1/4) = 7/16\end{aligned}$$

In  $F(2, 3, -2, 1)$  there are, in total, 16 positive numbers **non** uniformly distributed in  $[1/4, 7/2]$ .



$$1.00 \times 2^{-2} = 1 \ 1/4 = 1/4 = 4/16$$

$$1.01 \times 2^{-2} = (5/4) (1/4) = 5/16$$

$$1.10 \times 2^{-2} = (3/2) (1/4) = 3/8 = 6/16$$

$$1.11 \times 2^{-2} = (7/4) (1/4) = 7/16$$

$$1.00 \times 2^{-1} = 1 \ 1/2 = 1/2 = 4/8$$

$$1.01 \times 2^{-1} = (5/4) (1/2) = 5/8$$

$$1.10 \times 2^{-1} = (3/2) (1/2) = 3/4 = 6/8$$

$$1.11 \times 2^{-1} = (7/4) (1/2) = 7/8$$

$$1.00 \times 2^0 = 1 = 4/4$$

$$1.01 \times 2^0 = 5/4$$

$$1.10 \times 2^0 = 3/2 = 6/4$$

$$1.11 \times 2^0 = 7/4$$

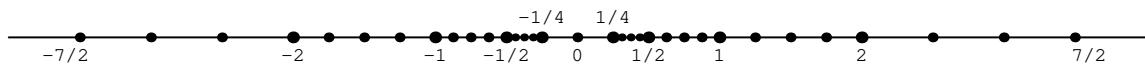
$$1.00 \times 2^1 = 1 \ 2 = 2 = 4/2$$

$$1.01 \times 2^1 = (5/4) 2 = 5/2$$

$$1.10 \times 2^1 = (3/2) 2 = 3 = 6/2$$

$$1.11 \times 2^1 = (7/4) 2 = 7/2$$

$F(2, 3, -2, 1)$  contains 33 finite numbers: 16 negative, zero, 16 positive



- They are non uniformly distributed.
- Their density decreases when the absolute value of the number increases.
- Within each interval  $[\beta^q, \beta^{q+1}]$ , they are in equal number and uniformly distributed.

## 2.5 Underflow and Overflow

If the non zero real number  $\alpha = \pm a_0 \bullet a_1 a_2 \dots a_t a_{t+1} \dots \times \beta^p$  is such that

$$p_{min} \leq p \leq p_{max}, \quad a_0 \neq 0, \quad a_i = 0 \quad \text{for } i \geq t$$

then  $\alpha \in F(\beta, t, p_{min}, p_{max})$ , i.e.  $\alpha$  is already a finite number.

If, instead,  $\alpha \notin F(\beta, t, p_{min}, p_{max})$ , then the problem arises on how to associate, in an adequate way, to  $\alpha$  a finite number  $fl(\alpha)$ .

There can be two cases of *not belonging*:

1. exponent  $p \notin [p_{min}, p_{max}]$ ;  
 if  $p < p_{min}$ , zero is associated to  $\alpha$  and the system, in general, signals an *underflow* situation;  
 if  $p > p_{max}$ ,  $\alpha$  is represented with a special symbol (NaN or Infinity) and the system, in general, signals an *overflow* situation;
2.  $p \in [p_{min}, p_{max}]$ , but  $\alpha \notin F(\beta, t, p_{min}, p_{max})$  because its digits  $a_i$  for  $i \geq t$  are **not** all zero;  
 in this case, a finite number  $fl(\alpha)$  is associated to  $\alpha$ , via chopping or rounding.

**Example 2.9** Consider  $F(10, 3, -4, 4)$ .

$$\begin{aligned}\alpha &= 98273 = 9.8273 \times 10^4, \\ fl_c(\alpha) &= 9.82 \times 10^4, \\ fl_r(\alpha) &= 9.83 \times 10^4.\end{aligned}$$

$$\begin{aligned}\alpha &= 99960 = 9.9960 \times 10^4, \\ fl_c(\alpha) &= 9.99 \times 10^4, \\ fl_r(\alpha) &= 10.0 \times 10^4 = 1.0 \times 10^5 \text{ overflow}.\end{aligned}$$

$$\begin{aligned}\alpha &= 0.00009995 = 9.995 \times 10^{-5}, \\ fl_c(\alpha) &= 9.99 \times 10^{-5} \text{ underflow}, \\ fl_r(\alpha) &= 10.0 \times 10^{-5} = 1.00 \times 10^{-4}.\end{aligned}$$

### 2.7.1 Special Representations

Zero is not the only number for which the IEEE standard has a special representation. Other special representation are introduced to handle *exceptional* situations, in order to give the possibility to continue the process during which such *exceptions* were generated.

- $\pm\infty$ ; in the case of a *division by zero*, for example, the result gets memorized as  $\infty$  and the process can continue; this avoids generating a message of *overflow* and ending.
- ‘Not a Number’ (NaN); it is not a number, but a pre-determined sequence (pattern) of bits that signals an error; e.g.  $0/0, \sqrt{-1}$ .
- non normalized (*subnormal*) numbers ; they allow the so-called *gradual underflow* by filling, in an uniform way, the gap between the first finite number to the left of zero and the first finite number to the right of zero; they are of the form:

$$fl(\alpha) = \pm 0.\bullet 00\dots 0 a_k a_{k+1} \dots a_{t-1} \times 2^{-p_{min}}.$$

## 2.8 Precision and Significant Digits

The number  $t$  of digits (binary digits = bits, in almost all modern computers) of the mantissa  $m_t$  corresponds to the **precision** with which we work and determines the amount of (decimal) **significant digits** with which a number gets represented:

$$|\log_{10} \beta^{-t}|$$

- In the IEEE *basic Single* format:

$$t = 23 + 1, \quad |\log_{10} 2^{-24}| \approx 7.22472$$

i.e. **single precision** corresponds to a finite representation with roughly 7 decimal significant digits.

- In the IEEE *basic Double*

$$t = 52 + 1, \quad |\log_{10} 2^{-53}| \approx 15.9546$$

i.e. **double precision** corresponds to a finite representation with roughly 16 decimal significant digits.

## 2.9 Machine Epsilon

$$\epsilon_{machine} = \beta^{1-t} \quad (2.9)$$

It is also called *Machine Zero* and it corresponds to the gap between the finite number 1 and its consecutive finite number. (which is the smallest finite number greater than 1).

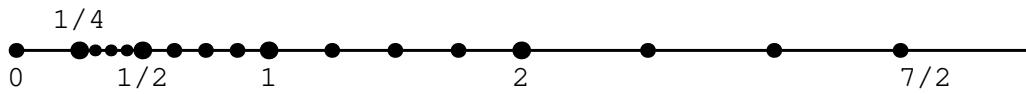


Figure 2.4: In the set  $F(2, 3, -2, 1)$  we have  $\epsilon_{machine} = 1/4$

**Example 2.10**

- IEEE *basic Single* format ( $t = 23 + 1$ ):

$$\epsilon_{\text{machine}} = \beta^{1-t} = 2^{-23} \approx 1.19 \times 10^{-7}$$

It is again confirmed that single precision corresponds to roughly 7 decimal significant digits.

- IEEE *basic Double* format ( $t = 52 + 1$ ):

$$\epsilon_{\text{machine}} = \beta^{1-t} = 2^{-52} \approx 2.22 \times 10^{-16}$$

It is again confirmed that double precision corresponds to roughly 16 decimal significant digits.

## 2.10 Rounding Unit

$$u = \begin{cases} \epsilon_{machine} & \text{chopping} \\ \frac{1}{2} \epsilon_{machine} & \text{rounding} \end{cases} \quad (2.10)$$

**Note.**  $u$  is the smallest positive finite number such that

$$fl(u + 1) > 1 .$$

- In the case of round-to-even,  $u$  is the smallest positive finite number such that  $fl(u + 1) = 1$ .

## 2.11 Absolute Error and Relative Error

Let  $fl(\alpha)$  be a finite number, with which we approximate the real number  $\alpha$ :

$$fl(\alpha) \approx \alpha.$$

**Definition 2.2 (Errors due to finite representation)**

$$\begin{aligned} E_{abs}(\alpha) &:= |\alpha - fl(\alpha)| && \text{absolute error} \\ E_{rel}(\alpha) &:= \frac{|\alpha - fl(\alpha)|}{|\alpha|} && \text{relative error} \end{aligned} \tag{2.11}$$

$E_{rel}$  is not defined if  $\alpha = 0$ . We then prefer the alternative definition

$$E_{rel}(\alpha) := |\delta| \quad \text{with } fl(\alpha) = \alpha(1 + \delta) \tag{2.12}$$

The two definitions are equivalent:

$$\frac{|\alpha - fl(\alpha)|}{|\alpha|} = \frac{|\alpha - \alpha(1 + \delta)|}{|\alpha|} = |\delta|$$

**Theorem 2.1 ( $E_{\text{rel}}$  and  $u$ )**

*This theorem gives an upper bound for the relative error, both with chopping and rounding.*

*For all real number  $\alpha \neq 0$ , it holds, both with chopping and rounding, that:*

$$\left| \frac{\alpha - fl(\alpha)}{\alpha} \right| \leq u. \quad (2.13)$$

**Corollary 2.1 ( $E_{\text{rel}}$  and  $u$ )**

*For all real number  $\alpha$ , it holds, both with chopping and rounding, that:*

$$fl(\alpha) = \alpha(1 + \delta), \quad |\delta| \leq u. \quad (2.14)$$

**Note.** Equivalent formulations:

$$fl(\alpha) = \alpha(1 - \delta), \quad fl(\alpha) = \frac{\alpha}{1 + \delta}, \quad fl(\alpha) = \frac{\alpha}{1 - \delta}, \quad |\delta| \leq u.$$

Indeed:  $\alpha \frac{1}{1 \mp \eta} = \alpha \left( 1 \pm \frac{1}{1 \mp \eta} \right)$

## 2. Elementi fondamentali del linguaggio

### ■ Riscrittura di termini

#### 2.1 Espressioni: espressioni normali (2.1.1)

#### 2.1 Espressioni: atomi (2.1.2)

Un atomo, in *Mathematica*, e' una espressione che non puo' essere suddivisa in espressioni piu' piccole.

⌘ Esistono fondamentalmente **tre** tipi di atomi: simbolo, numero, stringa di caratteri.

### ■ Simboli

### ■ Numeri

Esistono **quattro** tipi di numeri in **Mathematica**: interi, razionali, reali, complessi.

**Integer:** consiste di una sequenza di cifre decimali **ddddd**

Rational: **ha la forma** `intero1/intero2`

**Real:** ha la forma **ddd.ddd** ossia e' un numero a virgola mobile

**Complex:** ha la forma **a+bI** in cui **a, b** possono essere di qualsiasi dei tre tipi precedenti.

```
(* Integer *)
```

```
1 234 567 890
```

```
1 234 567 890
```

```
(* Rational *)
```

```
2/3
```

```
2  
—  
3
```

```
(* matNO=MatrixForm[{{1,2,3},{4,5,6}}];  
mat={{1,2,3},{4,5,6}};  
MatrixForm[mat] *)
```

```
(1 2 3)  
(4 5 6)
```

```
(* Real *)
nr = 21345.6789;
(* Di default , in output vengono mostrate 6 cifre significative *)
(* NumberForm puo' essere usato per
specificare quante cifre mostrare in output *)
{nr,
 NumberForm[nr, DefaultPrintPrecision → 9],
 FullForm[nr],
 Precision[nr],
 ScientificForm[nr],
 ScientificForm[nr, 9]} // TableForm
(* Il Tick nell'output di FullForm[nr] e' un NumberMarks e puo'
essere usato per specificare il numero di cifre significative *)

21345.7
21345.6789
21345.6789`MachinePrecision
2.13457 × 104
2.13456789 × 104
```

```
(* ScientificForm[nr] suggerisce
come riscrivere nr in forma esatta nint *)
(* Il punto decimale indica al
Kernel che il numero e' in MachinePrecision *)
nint = 123456789 × 10-4;
tt = {nint, 123456789. × 10-4, 123456789 × 10.-4, 123456789 × 10-4.}
Map[MachineNumberQ, tt]
```

$$\left\{ \frac{123456789}{10000}, 12345.7, 12345.7, 12345.7 \right\}$$

```
{False, True, True, True}
```

```
(* N numericizza *)
tt = {N[nint], N[nint, 20]}
Map[FullForm, tt]
(* N[nint,20] e' un BigNumber i.e. un numero
   a precisione arbitraria: qui ha precisione 20 *)
Map[MachineNumberQ, tt]

{12 345.7, 12 345.6789000000000000}
```

```
{12345.6789` , 12345.6789`20.}
```

```
{True, False}
```

```
(* Complex *)
(* Nota: commentare l'output di cc *)
cc = {2/3 + 4 I,
      2/3 + (45/10) I,
      2/3 + 4.5 I}
```

```
{ $\frac{2}{3} + 4i$ ,  $\frac{2}{3} + \frac{9i}{2}$ ,  $0.666667 + 4.5i$ }
```

⌘ Ognuno dei tipi numerici puo' avere virtualmente un numero di cifre illimitato.

⌘ L'esempio che segue e' in **aritmetica esatta**:

**Mathematica** suppone che l'input (intero) significhi che il calcolo deve essere fatto in modo esatto, per cui usa tante cifre quante sono necessarie per ottenere l'output esatto.

Paragoniamo gli esempi seguenti:

```
(* INTERO: Input esatto. Output esatto, qui ha 52 cifre decimali *)
(*  $\beta$  = base ,  $\eta$  = esponente *)
{ $\{\beta, \eta\} = \{5, 73\}$ ;
 out =  $\beta^\eta$ , Map[Precision, { $\beta, \eta, out$ }]]}
```

```
{1 058 791 184 067 875 423 835 403 125 849 552 452 564 239 501 953 125, {\infty, \infty, \infty}}
```

```
(* REALE: Un input in precisione macchina → Output in precisione macchina *)
 $\{\{\beta, \eta\} = \{5., 73\}; \text{out2} = \beta^\eta, \text{Map[Precision, } \{\beta, \eta, \text{out2}\}\}]$ 
(* la Machine Precision e' virale *)
 $\{\{\beta, \eta\} = \{5, 73.\};$ 
 $\text{out3} = \beta^\eta, \text{Map[Precision, } \{\beta, \eta, \text{out3}\}\}]$ 
 $\{1.05879 \times 10^{51}, \{\text{MachinePrecision}, \infty, \text{MachinePrecision}\}\}$ 
```

```
 $\{1.05879 \times 10^{51}, \{\infty, \text{MachinePrecision}, \text{MachinePrecision}\}\}$ 
```

```
(* REALE (precisione arbitraria , Big Number):
Qui, l' input a precisione piu' bassa ha (circa) 25 cifre decimali.
Output ha (circa) 23 cifre decimali *)
 $\eta = 73;$ 
(*  $\beta$  ha 24 zeri *)
 $\beta = 5.000000000000000000000000000000;$ 
 $\{\text{out4} = \beta^\eta, \text{Map[Precision, } \{\beta, \text{out4}\}\}]$ 
 $\{1.0587911840678754238354 \times 10^{51}, \{24.699, 22.8356\}\}$ 
```

⌘ La presenza del punto decimale, nell'input, e' interpretata come voler significare che l'input e' approssimato (non esatto) ed e' noto solo fino al numero di cifre che sono state esplicitamente scritte (fornite in input, appunto).

⌘ **Mathematica**, di conseguenza, esegue il calcolo in precisione arbitraria, mantenendo traccia di quante cifre nella risposta (output) sono giustificate dal numero di cifre significative nell'input & nelle operazione che sono state svolte.

Nell'esempio appena visto sono state perse le 2 cifre meno significative (si e' partiti dalla precisione di circa 25 e si e' arrivati ad una precisione di circa 23):

```

(* Precision fornisce il numero di
cifre significative in un numero approssimato.

NB. Il risultato di Precision puo' non corrispondere
esattamente al numero di cifre mostrato nella cella.

Il motivo e' che la precisione numerica viene
calcolata in base binaria e poi convertita in decimale. *)

 $\beta = 5.00000000000000000000000000000000$ ;  $\eta = 73$ ; out4 =  $\beta^{\eta}$ ;
{Precision[ $\beta$ ], Precision[ $\beta^{\eta}$ ]}
```

{24.699, 22.8356}

⌘ Si puo' specificare la precisione di un numero approssimato, esplicitamente, usando la sintassi **numero`precision** oppure usando N[ ]

```

 $\eta = 73;$ 
(* Input con 25 cifre decimali . Output, qui, ha (circa) 24 cifre decimali *)
 $\beta = 5^{`25}; \{out5 = \beta^{\eta}, \text{Map[Precision, } \{ \beta, out5 \} \}\}$ 

 $\beta = N[5, 25];$ 
 $\{out6 = \beta^{\eta}, \text{Map[Precision, } \{ \beta, out6 \} \]\}$ 

{1.05879118406787542383540 × 1051, {25., 23.1367}}

```

---

```

{1.05879118406787542383540 × 1051, {25., 23.1367}}

```

⌘ I numeri approssimati, che vengono dati in input con un numero di cifre **non** superiore a quello messo a disposizione dall'hardware a virgola mobile del computer, vengono memorizzati in formato, nativo, in virgola mobile a doppia precisione.

Tutte le operazioni su tali numeri sono eseguite in hardware.

Tali numeri sono detti **a precisione di macchina**.

`$MachinePrecision`

15.9546

⌘ La variabile (read-only) di sistema \$MachinePrecision specifica la precisione dei numeri a virgola mobile nativi, che puo' variare su differenti architetture.

⌘ La funzione MachineNumberQ[ ] puo' essere usata per determinare se un numero approssimato e' un numero macchina

```
MachineNumberQ[5.0000000000000000000000000]
(* Questo input ha 24 zeri,
quindi ha troppe cifre per poter essere un numero macchina *)
False
```

```
MachineNumberQ[5.0]
(* Questo input ha, implicitamente,
un numero di cifre significative pari a $MachinePrecision *)
True
```

```
5.^73
(* Questo calcolo e' svolto in hardware a virgola mobile *)
1.05879 × 1051
```

⌘ **Mathematica** fa vedere (displays) solo le prime poche cifre di un numero a precisione macchina, a meno che non venga richiesto diversamente.

Per vedere tutte le cifre, possiamo usare **FullForm[]**.

⌘ **FullForm[]** stampa la forma interna di qualsiasi cosa presente nell'espressione

```
{5.^73, FullForm[5.^73]}

{1.05879 × 1051, 1.0587911840678756`*^51}
```

```
Sqrt[2] + a
FullForm[Sqrt[2] + a] (* Sqrt[2] e' 2^(1/2) *)

 $\sqrt{2} + a$ 
```

```
Plus[Power[2, Rational[1, 2]], a]
```

```
Sqrt[2.] + a
FullForm[Sqrt[2.] + a]

1.41421 + a

Plus[1.4142135623730951` , a]
```

```
FullForm[Plot[x^2, {x, 0, 1}]];
```

In alternativa, si puo' usare InputForm[].

$\text{\&#8804;}$  InputForm[ ] fa vedere (displays) cio' che si deve scrivere in input per ottenere qualcosa di uguale alla espressione data

```
InputForm[5.^73]
```

```
1.0587911840678756*^51
```

```
InputForm[Sqrt[2] + a]
```

```
Sqrt[2] + a
```

```
InputForm[Sqrt[2.] + a]
```

```
1.4142135623730951 + a
```

? InputForm

Symbol

i

InputForm[*expr*] prints as a version of *expr* suitable for input to the Wolfram Language.

▼

### Nota.

La precisione di un numero macchina e' sempre \$MachinePrecision.

Al contrario, se la precisione di un numero arbitrario e' uguale od inferiore a \$MachinePrecision, tale numero e' comunque memorizzato internamente come numero a precisione arbitraria.

Conviene, pertanto, usare MachineNumberQ per essere sicuri.

```
Map[MachineNumberQ,
{N[1/3, 5],
 N[1/3, 15],
 N[1/3, $MachinePrecision],
 N[1/3, MachinePrecision],
 N[1/3]}]
```

```
{False, False, False, True, True}
```

```
(* NB. $MachinePrecision e' il valore numerico del simbolo MachinePrecision *)
{FullForm[$MachinePrecision],
 FullForm[MachinePrecision]}
{(* Equal *) $MachinePrecision == MachinePrecision,
 (* SameQ *) $MachinePrecision === MachinePrecision}

{15.954589770191003` , MachinePrecision}

{True, False}
```

```
(* NOTE abbreviate su Equal, SameQ *)
(* SameQ: numeri Real , diversi nell'ultimo bit, sono considerati identici *)
(* Equal: numeri approssimati, a precisione macchina o maggiore,
sono considerati uguali se differiscono
negli ultimi 7 bit i.e. ultime 2 cifre decimali *)
(* Equal:
usa approssimazioni numeriche per stabilire la uguaglianza tra numeri esatti *)
```

```
(* NOTE. Equal puo' restituire , in output,
l'input UnEvaluated . SameQ restituisce sempre un booleano *)
x == y
x === y
```

x == y

False

### Nota.

Un input che contiene un punto decimale e' sempre considerato essere un numero approssimato, anche quando noi sappiamo che non lo e'.

D'altra parte **Mathematica** non puo' fare ipotesi diverse, altrimenti si incorrerebbe in errori:

```
(* Il risultato di questo calcolo e' zero approssimato *)
3/4 - 0.75
Precision[%]

0.

MachinePrecision
```

Se nell'esempio qui sopra avessimo inteso scrivere 0.75 (ossia "tre quarti") in modo esatto, allora avremmo

dovuto dare in input  $75/100$  (che viene ridotto a  $3/4$ ):

**75 / 100**

$\frac{3}{4}$

**3 / 4 – 75 / 100**

**Precision[%]**

0

$\infty$

### Per riassumere.

Ci sono due classi ampie di numeri:

**esatti**, che includono interi, razionali, complessi con coefficienti esatti;

**approssimati**, fatti da numeri che contengono sempre la virgola mobile.

⌘ I numeri **approssimati** sono suddivisi in due sottoclassi:

a precisione di **macchina**;

a precisione **arbitraria**.

⌘ **Mathematica** segue una convenzione inusuale per maneggiare i numeri esatti.

Di default, **Mathematica** non esegue mai una (qualsiasi) operazione numerica che potrebbe convertire una espressione esatta in una approssimata.

Per esempio:

**Sqrt[3]**

$\sqrt{3}$

**Mathematica** non riscrive l'espressione esatta  $\text{Sqrt}[3]$  come un numero, perche' per fare cio' dovrebbe inserire una approssimazione (dato che  $\text{Sqrt}[3]$  non ha una rappresentazione decimale finita).

⌘ D'altra parte, **Mathematica** valuta invece:

**Sqrt[3.]**

1.73205

L'argomento 3. (della funzione  $\text{Sqrt}[ ]$ ) e' considerato approssimato (perche' contiene la virgola mobile).

Dato che il numero 3. e' gia' approssimato, **Mathematica** non ha alcun problema a calcolare la sua radice quadrata in modo approssimato.

⌘ Possiamo chiedere a *Mathematica* di valutare numericamente ogni espressione esatta, usando la fun-

zione N (come già visto):

```
enne = N[Sqrt[3]]  
(* Di default, il risultato e' un numero macchina *)  
Precision[enne]  
{FullForm[enne], InputForm[enne]};  
  
1.73205
```

1.73205

## MachinePrecision

⌘ Un secondo argomento (opzionale) alla funzione N specifica la precisione desiderata nel risultato (come già visto):

```
enne20 = N[Sqrt[3], 20]
Precision[enne20]

1.7320508075688772935
```

20.

⌘ In generale, se anche solo un degli argomenti (passati ad una funzione built-in numerica) e' approssimato, la funzione verra' valutata, convertendo tale argomento a numero approssimato, nella precisione piu' alta giustificabile.

Rivediamolo con un esempio:

```
(* Il risultato e' un numero macchina *)
{add = 1 + 2.5, Precision[add]}
{3.5, MachinePrecision}
```

31.3979

```
(* NB. Cambiamo il primo addendo,
per creare un caso di Cancellazione Numerica *)
canc = (-25 / 10) + addendo;
{canc // Chop, Precision[canc]}

{0, 0.}
```

31.

$$\{-9.000000000000000 \times 10^{-9}, 22.9542\}$$

⌘ Un buon riferimento e' il tutorial Numbers

```
Hyperlink[Framed["Tutorial sui Numeri"],  
 "https://reference.wolfram.com/language/tutorial/Numbers.html"]
```

Tutorial sui Numeri

- Stringhe di caratteri
  - Esercizio 1S pg. 255

## 2.2 Valutazione di espressioni

Il processo di valutazione (Evaluation) di base e' semplice.

⌘ Il Kernel continua a riscrivere termini fino a che non rimane nulla che esso sappia riscrivere in una forma diversa.

⌘ Dato che la riscrittura di termini (term rewriting) rimpiazza una espressione con un'altra, qualsiasi cosa sia rimasta (quando tale processo termina) deve essere una espressione valida: questo implica che l'insieme di tutte le espressioni è *chiuso* rispetto alla valutazione.

⌘ Questo permette che ogni espressione sia annidata dentro una qualsiasi altra (anche se il risultato del fare cio' potrebbe non avere senso ;-).

<sup>⌘</sup> In analogia alla chiamata di funzione in altri linguaggi, chiamiamo *valore di ritorno* (return value) di un'applicazione.

## 2. Elementi fondamentali del linguaggio

### ■ Riscrittura di termini

#### 2.1 Espressioni: espressioni normali (2.1.1)

#### 2.1 Espressioni: atomi (2.1.2)

#### 2.2 Valutazione di espressioni

Il processo di valutazione (Evaluation) di base e' semplice.

⌘ Il Kernel continua a riscrivere termini fino a che non rimane nulla che esso sappia riscrivere in una forma diversa.

⌘ Dato che la riscrittura di termini (term rewriting) rimpiazza una espressione con un'altra, qualsiasi cosa sia rimasta (quando tale processo termina) deve essere una espressione valida: questo implica che l'insieme di tutte le espressioni e' *chiuso* rispetto alla valutazione.

⌘ Questo permette che ogni espressione sia annidata dentro una qualsiasi altra (anche se il risultato del fare cio' potrebbe non avere senso ;-).

⌘ In analogia alla chiamata di funzione in altri linguaggi, chiamiamo *valore di ritorno* (return value) di una data espressione il risultato della valutazione di tale espressione.

*Trace[ ]*

E' possibile ottenere una descrizione *post mortem* della valutazione di una qualsiasi espressione, utilizzandola come argomento di *Trace[]* i.e. impacchettando tale espressione dentro la head *Trace*.

(\*N.B. Se usassimo *TreeForm* qui,  
*TreeForm[Sin[Log[2.5,7]]]* restituisce solo il risultato,  
che e' un numero a precisione macchina \*)

**Trace[Sin[Log[2.5, 7]]]**

{Log[2.5, 7], 2.12368}, Sin[2.12368], 0.851013}

### ? Log

Symbol

i

$\text{Log}[z]$  gives the natural logarithm of  $z$  (logarithm to base e).

$\text{Log}[b, z]$  gives the logarithm to base  $b$ .

▼

(\*  $\text{Log}[a,b]$  viene riscritto come  $\text{Log}[b]/\text{Log}[a]$ \*)

$\text{Log}[a, b] == \text{Log}[E, b]/\text{Log}[E, a] == \text{Log}[b]/\text{Log}[a]$

$\text{Trace}[\text{Log}[a, b]];$

True

**Nota.** I passi della valutazione di  $\text{Sin}[\text{Log}[2.5, 7]]$  sono :

1a.  $\text{Log}[2.5, 7]$  viene riscritto come  $\text{Log}[7]/\text{Log}[2.5]$

1b.  $\text{Log}[7]$  viene valutato numericamente

1c.  $\text{Log}[2.5]$  viene valutato numericamente

1. Il quoziente dei due risultati precedenti viene valutato numericamente.

2. Il  $\text{Sin}$  del quoziente precedente viene valutato numericamente

→ Il risultato è un atomo: è un numero reale che non può essere ulteriormente riscritto.

Il processo, pertanto, termina.

(\* 1a \*)  $\text{Log}[2.5, 7] == \text{Log}[7]/\text{Log}[2.5]$

(\* 1b \*)  $\{\text{Log}[7], \text{Log}[7.]\}$

(\* 1c \*)  $\text{Log}[2.5]$

(\* 1 \*)  $\text{Log}[7.]/\text{Log}[2.5]$

(\* 2 \*)  $\text{Sin}[%]$

True

{ $\text{Log}[7]$ , 1.94591}

0.916291

2.12368

0.851013

### Valutazione standard (e non-standard)

⌘ L'esempio appena visto mostra un punto importante del processo di valutazione.

In generale, le parti di una espressione normale vengono valutate prima dell'intera espressione.  
 Questo modo di procedere e' detto **valutazione standard**.

⌘ Gli argomenti di certe funzioni di **Mathematica**, al contrario, non vengono valutati prima che la funzione sia invocata (called).

Questo modo di procedere e' detto **valutazione non-standard**.

Ne vedremo un esempio piu' avanti.

⌘ In termini informatici, una espressione e' un albero (tree) e la sua valutazione viene eseguita "depth-first" i.e. vengono valutate per prime le parti (sotto-espressioni) che stanno a maggiore profondita' nell'albero rappresentante l'espressione stessa (foglie).

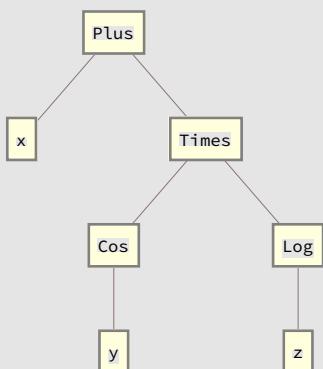
⌘ Le parentesi graffe indicano il livello di profondita' (Depth) della sotto-espressione correntemente valutata. L'annidamento delle parentesi graffe diventa maggiore via via che il processo di valutazione si addentra nell'espressione, fino a raggiungerne le foglie.

Viceversa, l'annidamento delle parentesi graffe diventa minore via via che il processo di valutazione risale nell'espressione, tornando indietro verso la sua radice (Head).

⌘ Possiamo usare `TreeForm[ ]` per stampare una espressione, esplicitamente, in forma di albero (con le eventuali limitazioni imposte dall'output ASCII).

L'output della `TreeForm[]` puo' risultare non troppo leggibile, specie per espressioni molto grandi ed articolate. In tale caso, e' meglio usare `FullForm[ ]`, studiandola attentamente.

```
TreeForm[x + Cos[y] Log[z], ImageSize → Small]
```



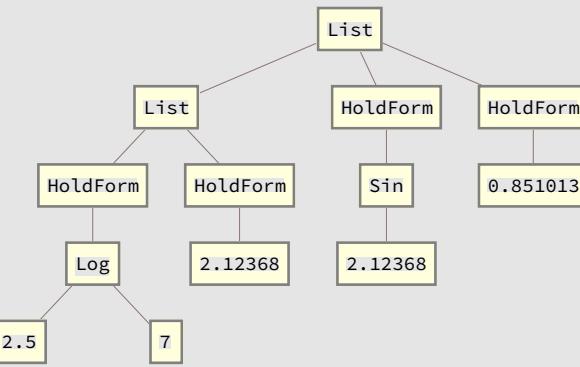
### **HoldForm[]**

Se applichiamo `TreeForm` alla espressione `Trace[ Sin[Log[2.5,7]] ]` dell'esempio precedente, nell'albero (rappresentante tale espressione) appare `HoldForm[ ]`.

`HoldForm[expr]` restituisce la stampa di una espressione, mantenendo tale espressione in formato non valutato.

```
Trace[ $\text{Sin}[\text{Log}[2.5, 7]]$ ]
Trace[ $\text{Sin}[\text{Log}[2.5, 7]]$ ] // TreeForm
```

```
{ $\{\text{Log}[2.5, 7], 2.12368\}, \text{Sin}[2.12368], 0.851013$ }
```



⌘ Nel nostro esempio, HoldForm[ ] (usata da Trace[ ]) serve per stampare i risultati intermedi (messi in evidenza da Trace[ ]) come se fossero non valutati.

⌘ Se HoldForm[ ] **non** fosse presente nei risultati intermedi, otterremmo direttamente la TreeForm dell'atomo rappresentante il risultato della valutazione complessiva.

Questo accade proprio perche' (senza HoldForm) l'argomento di TreeForm[ ] viene valutato a numero reale **prima** che la TreeForm stessa venga valutata.

Lo stesso vale per FullForm.

Vediamolo:

```
(* HoldForm, usata da Trace, serve per stampare output intermedi di Trace.
Senza HoldForm, otteniamo TreeForm dell'atomo rappresentante l'output finale,
risultante dalla valutazione complessiva *)
espressione = Sin[Log[2.5, 7]];
TreeForm[espressione, ImageSize → Small]
FullForm[espressione]
```

```
0.851013
```

```
0.851012661490406`
```

⌘ Per vedere la struttura interna di un'espressione, la cui valutazione complessiva restituisce un numero (e.g.  $\text{Sin}[\text{Log}[2.5, 7]]$ ), e' necessario inibirne la valutazione.

Per fare cio', possiamo impacchettare tale espressione dentro una head che impedisca ai suoi argomenti di essere valutati .

Un esempio di tale head e' HoldForm[ ] .

⌘ HoldForm[ ] inibisce la valutazione dei suoi argomenti. In questo modo, permette di vedere esplicitata la struttura di un'espressione (cui HoldForm[ ] sia stata applicata).

⌘ HoldForm[] realizza, pertanto, una valutazione non-standard

### ? HoldForm

Symbol

*i*

HoldForm[*expr*] prints as the expression *expr*, with *expr* maintained in an unevaluated form.

▼

⌘ Un buon riferimento e' il tutorial **Evaluation**

```
Hyperlink[Framed["Tutorial su Evaluation"],
 "https://reference.wolfram.com/language/tutorial/Evaluation.html"]
```

Tutorial su Evaluation

### Attributes[]

Per verificare se una Head simbolica inibisce (o meno) la valutazione delle sue parti, possiamo esaminarne le caratteristiche, con la built-in Attributes[ ]

#### Attributes[HoldForm]

```
(* HoldForm ha la caratteristica HoldAll i.e. per tutte le
   parti incluse nell'head HoldForm[] e' inibita la valutazione *)
(* HoldForm ha la caratteristica Protected
   i.e. il suo nome non e' ridefinibile da utente *)
```

```
{HoldAll, Protected}
```

Un punto di attenzione (ancora su Trace, HoldForm e attributo HoldAll)

### HoldForm vs Hold

⌘ Come detto in precedenza, il procedimento di valutazione continua fino a che non rimane piu' nulla che possa essere riscritto in un'altra forma.

Se non ci fossero head come HoldForm, non ci sarebbe modo di ottenere il risultato di una valutazione parziale di una espressione (quale e', appunto, un componente di una Trace[ ]).

# First Look at Lists

A **List** is a basic way to collect or store things together.

`{1, 2, 3}` is a list of numbers.

`{..., ...}` is the special form of `List[..., ...]`

Unlike, say, Plus, the function **List** does not actually compute anything.

So, if you give a list as input, it will just come back unchanged.

```
testList = {1, 2, 3, 4, a, b, c}
FullForm[testList]

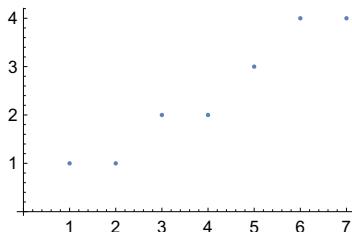
heterogeneousList = {1, 1/2, 2/3., Pi, a, Graphics[Circle[]]}

integerList = {1, 1, 2, 3, 4}

?Intersection
```

**ListPlot** is a function that makes a plot of a **List** of numbers.

```
myList = {1, 1, 2, 2, 3, 4, 4};
ListPlot[myList, ImageSize → Small ]
```



ListPlot plots the values of subsequent list elements, i.e. points (X, Y) :

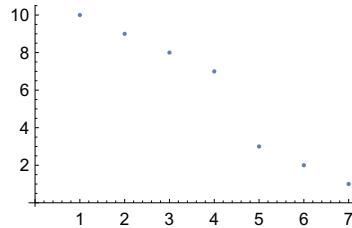
the X value gives the **position in the list** (default is `{1, 2, 3, 4, 5, 6, 7}....`) ;

the Y value gives the **value** of that element.

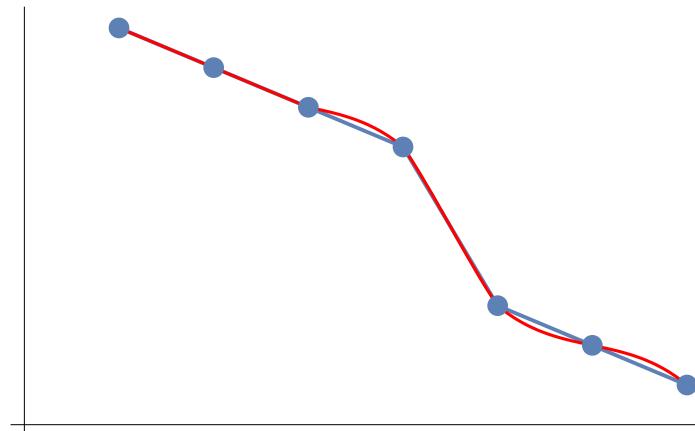
```
(* myList={1,1,2,2,3,4,4}; *)
ListPlot[myList] == ListPlot[{{1, 1}, {2, 1}, {3, 2}, {4, 2}, {5, 3}, {6, 4}, {7, 4}}]
True

anotherList = {10, 9, 8, 7, 3, 2, 1};
ListPlot[anotherList] == ListPlot[{{1, 10}, {2, 9}, {3, 8}, {4, 7}, {5, 3}, {6, 2}, {7, 1}}]
True
```

```
(* anotherList={10,9,8,7,3,2,1}; *)
ListPlot[anotherList, ImageSize → Small ]
```

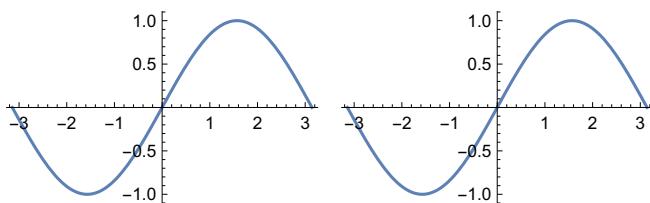


```
(* anotherList={10,9,8,7,3,2,1}; *)
lpoints = ListPlot[anotherList, PlotStyle → PointSize[0.03]];
lp = ListPlot[anotherList,
    Joined → True,
    PlotStyle → Thick];
interpFun = Interpolation[anotherList];
ip = Plot[interpFun[t], {t, 1, 7}, PlotStyle → Red];
Show[lp, ip, lpoints, Ticks → None]
```

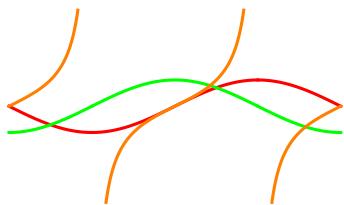


```
(* Display two plots with GraphicsRow *)
```

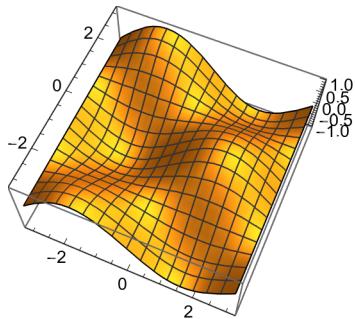
```
GraphicsRow[
{Plot[ Sin[θ] , {θ, -Pi, Pi}],
 Plot[ {Sin[θ]} , {θ, -Pi, Pi}]},
 ImageSize → Medium
]
```



```
(* Display three functions with one Plot *)
Plot[ {Sin[θ], Cos[θ], Tan[θ]},
  {θ, -Pi, Pi},
  PlotStyle → {Red, Green, Orange} ,
  Axes → None,
  ImageSize → Small ]
```



```
(* Display a 3D function *)
Plot3D[ Sin[θ] Cos[α],
  {θ, -Pi, Pi},
  {α, -Pi, Pi},
  ImageSize → Small ]
```

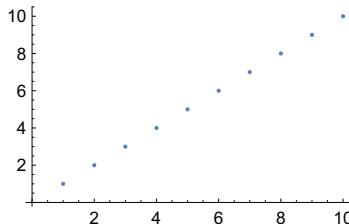


```
(* → Rule[ ] *) (* :> RuleDelayed[ ] *)
(* = Set[ ] *) (* := SetDelayed[ ] *)
(* Import[], Export[] *)

$ImportFormats
(* $ExportFormats *)
```

**Range** is a function that creates a **List** of numbers.

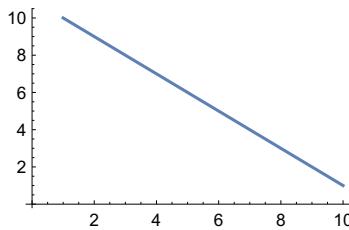
```
myRange = Range[10]
ListPlot[myRange, ImageSize → Small]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```



**Reverse** reverses the elements in a **List**.

```
(* myRange=Range[10]; *)
Reverse[myRange]

(* myRange=Range[10]; *)
reversed = Reverse[myRange];
ListPlot[reversed,
 Joined → True,
 ImageSize → Small]
```



**Join** joins lists together and creates a single **List** .

```
(* Elementes are unsorted *)
Join[{4, 5}, {1, 2, 3}, {6, 7}]
(* Duplicates are kept *)
Join[{1, 2, 3}, {1, 2, 3, 4, 5}]
{4, 5, 1, 2, 3, 6, 7}
{1, 2, 3, 1, 2, 3, 4, 5}

(* Union : duplicates are eliminated and sorted *)
Union[{1, 2, 3}, {1, 2, 3, 4, 5}]
{1, 2, 3, 4, 5}

ListPlot[
 Join[Range[20], Range[20], Range[30]],
 ImageSize → Small]

ListPlot[
 Join[Range[20], Reverse[Range[20]], Range[30]],
 ImageSize → Small]
```

---

## Vocabulary

{1,2,3,4}	list of elements
ListPlot[{1,2,3,4}]	plot a list of numbers
Range[10]	range of numbers
Reverse[{1,2,3}]	reverse a list
Join[{4,5,6},{2,3,2}]	join lists together (elements unsorted, duplicates kept)
Intersection	
Union	
Table	
Array	
Plot, Plot3D	
Show	
Import[], Export[]	
\$ImportFormats, \$ExportFormats	
Set (=), SetDelayed (:=)	
Equal ( == ), SameQ ( === )	
Function (pure function)	
Rule ( → )	
Clear	
Interpolation	
NSolve	

---

## Exercises

**3.1** Use Range to create the list {1, 2, 3, 4}.

**3.2** Make a list of numbers up to 100.

**3.3** Use Range and Reverse to create {4, 3, 2, 1}.

**3.4** Make a list of numbers from 1 to 50 in reverse order.

**3.5** Use Range, Reverse and Join to create {1, 2, 3, 4, 4, 3, 2, 1} .

**3.6** Plot a list that counts up from 1 to 100, then down to 1.

**3.7** Use Range and RandomInteger to make a list with a random length up to 10.

**3.8** Find a simpler form for Reverse[Reverse[Range[10]]].

**3.9** Find simpler forms for  $\text{Join}[\{1, 2\}, \text{Join}[\{3, 4\}, \{5\}]]$ .

```
task = Join[{1, 2}, Join[{3, 4}], {5}];
(* attenzione all'ordine *)
taskno = Join[{1, 2}, Join[{5}, {3, 4}]];
Range[5] == task
taskno == task

True
False
```

**3.10** Find a simpler form for  $\text{Join}[\text{Range}[10], \text{Join}[\text{Range}[10], \text{Range}[5]]]$ .

**3.11** Find a simpler form for  $\text{Reverse}[\text{Join}[\text{Range}[20], \text{Reverse}[\text{Range}[20]]]]$  ( $\text{PalindromeQ}$ ).

**+3.1** Compute the reverse of the reverse of  $\{1, 2, 3, 4\}$ .

**+3.2** Use Range, Reverse and Join to create the list  $\{1, 2, 3, 4, 5, 4, 3, 2, 1\}$ .

**+3.3** Use Range, Reverse and Join to create  $\{3, 2, 1, 4, 3, 2, 1, 5, 4, 3, 2, 1\}$ .

**+3.4** Plot the list of numbers  $\{10, 11, 12, 13, 14\}$ .

**+3.5** Find a simpler form for  $\text{Join}[\text{Join}[\text{Range}[10], \text{Reverse}[\text{Range}[10]]], \text{Range}[10]]$ .

## Tech Notes

### Syntax of Range

`Range[m, n]` generates numbers from **m** to **n**.

`Range[m, n, s]` generates numbers from **m** to **n** in steps of **s**.

```
Range[2, 9];
Range[2, 9, 1];
Range[2, 9, 3];
Range[2, 9, 3.];
N[Range[2, 9, 3]];
Range[2, 9, 1/2]
{2,  $\frac{5}{2}$ , 3,  $\frac{7}{2}$ , 4,  $\frac{9}{2}$ , 5,  $\frac{11}{2}$ , 6,  $\frac{13}{2}$ , 7,  $\frac{15}{2}$ , 8,  $\frac{17}{2}$ , 9}
```

**Range[]** forms a list from a range of numbers or other objects

**Table[]** makes a table of values of an expression (uses an iterator)

```
Table[x, {x, 2, 9, 1/2}]
Table[x, {x, 2, 9, 3}];
N[Table[x, {x, 2, 9, 3}]];
Table[x, {x, 2, 9, 3.}];
{2,  $\frac{5}{2}$ , 3,  $\frac{7}{2}$ , 4,  $\frac{9}{2}$ , 5,  $\frac{11}{2}$ , 6,  $\frac{13}{2}$ , 7,  $\frac{15}{2}$ , 8,  $\frac{17}{2}$ , 9}

(*Solvers for non linear equations : Solve and NSolve *)
sol = Solve[x^2 - 2 x + 1 == 0, x]
N[sol]
NSolve[x^2 - 2 x + 1 == 0, x]
```

## Lists (arrays) in other computer languages

Many computer languages have constructs like lists (often called "arrays"). Usually, though, they only allow lists of explicit things, like numbers; you cannot have a list like **{a, b, c}** if you have not said what **a, b, c** are. You can in *Mathematica* because it is symbolic.

```
{1, 1/2, Pi, a, Graphics[{Blue, Circle[]}], ImageSize -> Tiny} }

Array[] creates an array by applying a function f to successive indices

Array[f, 10]
(* Pure function *)
Union[
  Array[##+1 &, 8],
  Array[##+3/2 &, 7]
]
{2,  $\frac{5}{2}$ , 3,  $\frac{7}{2}$ , 4,  $\frac{9}{2}$ , 5,  $\frac{11}{2}$ , 6,  $\frac{13}{2}$ , 7,  $\frac{15}{2}$ , 8,  $\frac{17}{2}$ , 9}

(* Plus is Listable
 i.e. it is automatically threaded over lists that appear as its argument *)
Union[
  Array[## &, 8]+1,
  Array[## &, 7]+3/2
]
{2,  $\frac{5}{2}$ , 3,  $\frac{7}{2}$ , 4,  $\frac{9}{2}$ , 5,  $\frac{11}{2}$ , 6,  $\frac{13}{2}$ , 7,  $\frac{15}{2}$ , 8,  $\frac{17}{2}$ , 9}
```

```
(* SetDelayed := for function definition *)
a = 3;
f[x_] := x + 1
g[x_] := x + 3/2
Union[
  Array[f, 8],
  Array[g, 7]
]
{2,  $\frac{5}{2}$ , 3,  $\frac{7}{2}$ , 4,  $\frac{9}{2}$ , 5,  $\frac{11}{2}$ , 6,  $\frac{13}{2}$ , 7,  $\frac{15}{2}$ , 8,  $\frac{17}{2}$ , 9}

TreeForm[f[x]];
```

## Ordered List

{ a, b, c } is a list of elements in a definite order;

{ b, c, a } is a different list ;

Here, **Equal[]** returns Unevaluated , as it does not have information to establish equality (a, b, c are unassigned).

Instead, **SameQ[]** always returns True/False

```
Clear[a]

{a, b, c} == {b, c, a}
{a, b, c} === {b, c, a}
{3, b, c} == {b, c, 3}

False

(* === is the special form of SameQ *)
(* SameQ tests syntactic equality *)
(* SameQ requires exact correspondence between expressions, except that it
   considers Real numbers equal if they differ in their last binary digit *)

{1, 2, 3} == {3, 2, 1}
{1, 2, 3} === {3, 2, 1}

False

False

(* == is the special form of Equal *)
(* Equal tests mathematical equality *)
(* lhs == rhs returns True/False if lhs, rhs are numerically equal/unequal ,
   and it returns unevaluated if equality cannot be established *)
```

## Note on Equal and SameQ

(\* See § Background & Context in the help-page of Equal \*)

```
Reverse[Reverse[{x}]] === {x}  
Equal[{x}]  
SameQ[{x}]
```

# Displaying Lists

ListPlot is one way to display, or visualize, a list of numbers.

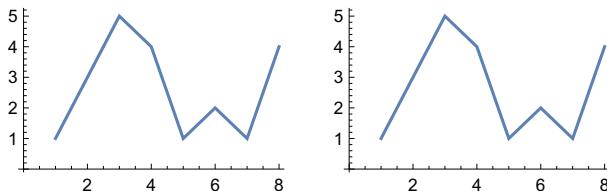
There are lots of others.

Different ones tend to emphasize different features of a list.

## ListLinePlot plots a list, joining up values

When values jump around, it is usually easier to understand if you join them up.

```
llp = ListLinePlot[{1, 3, 5, 4, 1, 2, 1, 4}, ImageSize → Small];
lp = ListPlot[{1, 3, 5, 4, 1, 2, 1, 4}, Joined → True, ImageSize → Small];
GraphicsRow[{llp, lp}]
(* ListLinePlot and ListPlot differ in PointSize *)
(* ListLinePlot uses Rational[7,360] ≈ 0.01944444444444445` *)
(* List Plot uses 0.01283333333333334` ≈ Rational[77, 6000]*)
llpR = ListLinePlot[{1, 3, 5, 4, 1, 2, 1, 4}, ImageSize → Small] /.
  Rational[7, 360] → 0.01283333333333334`;
{llp == lp, llpR == lp}
(* /. ReplaceAll *)
(* a+b /. a→ c *)
(* d+b /. a→ c *)
```

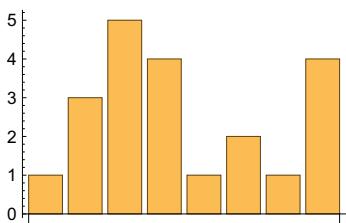


{False, True}

## Making a BarChart can be useful too

Values give bar heights:

```
BarChart[{1, 3, 5, 4, 1, 2, 1, 4}, ImageSize → Small]
```



## If the list is not too long, a **PieChart** can be useful

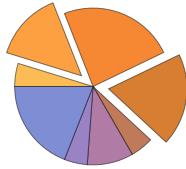
Values give the size of each slice (wedge).

Click on a slice to “explode” it.

Slices have a relative sizes determined by the relative sizes of numbers in the list.

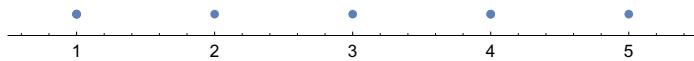
The slice for the first number starts at the 9 o’clock position; subsequent slices read clockwise.

```
PieChart[{1, 3, 5, 4, 1, 2, 1, 4}, ImageSize → Tiny]
```



## If you just want to know which numbers appear, you can **Plot** them on a **Number Line**

```
NumberLinePlot[{1, 3, 5, 4, 1, 2, 1, 4}]
(* Duplicates are eliminated and ordinates are sorted *)
Union[{1, 3, 5, 4, 1, 2, 1, 4}]
```



```
{1, 2, 3, 4, 5}
```

```
NumberLinePlot[{1, 7, 11, 25}];
```

## You may just want to put the elements of a list in a **Column**

```
Column[{1, 3, 5, 4, 1, 2, 1, 4}];
Column[{100, 350, 502, 400}]
100
350
502
400
```

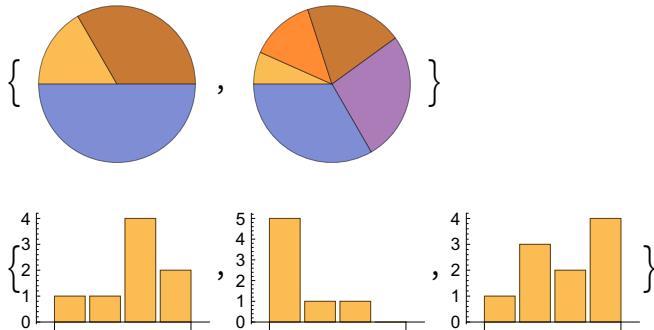
## Combine plots

- Lists can contain anything, including Graphics.

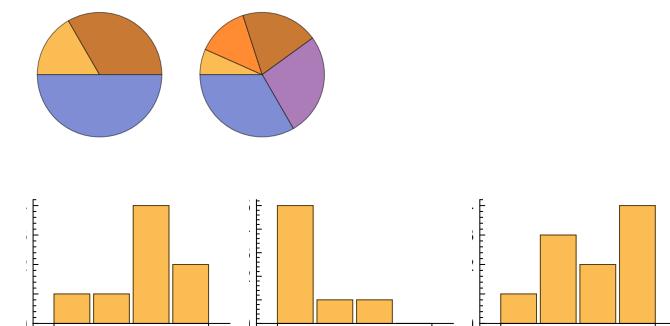
So, you can combine plots by putting them in lists.

- A list of plots can appear as the (input or) output of a computation, since *Mathematica* is symbolic.

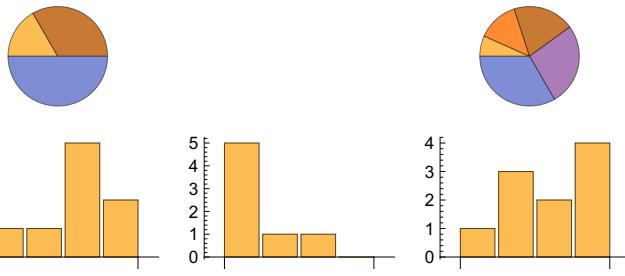
```
{PieChart[Range[3], ImageSize → Tiny], PieChart[Range[5], ImageSize → Tiny]}
{BarChart[{1, 1, 4, 2}, ImageSize → Tiny],
 BarChart[{5, 1, 1, 0}, ImageSize → Tiny], BarChart[{1, 3, 2, 4}, ImageSize → Tiny]}
```



```
(* To group graphics or images,
it is better to use GraphicsRow or GraphicsGrid *)
GraphicsRow[{PieChart[Range[3]], PieChart[Range[5]]}, ImageSize → Small]
GraphicsRow[
 {BarChart[{1, 1, 4, 2}], BarChart[{5, 1, 1, 0}], BarChart[{1, 3, 2, 4}]}, ImageSize → Medium]
```



```
(* Alignments in GraphicsGrid *)
GraphicsGrid[{
  {PieChart[Range[3], ImageSize → Tiny], " " ,
   PieChart[Range[5], ImageSize → Tiny]},
  {BarChart[{1, 1, 4, 2}, ImageSize → Small],
   BarChart[{5, 1, 1, 0}, ImageSize → Small],
   BarChart[{1, 3, 2, 4}, ImageSize → Small]}
}]
```



**Attributes[Plus]**

## Vocabulary

ListLinePlot[{1,2,5}]	values joined by a line
BarChart[{1,2,5}]	bar chart (values give bar heights)
PieChart[{1,2,5}]	pie chart (values give wedge sizes)
NumberLinePlot[{1,2,5}]	numbers arranged on a line
Column[{1,2,5}]	elements displayed in a column
GraphicsGrid	
TableForm	
ReplaceAll	
Attributes	

## Exercises

- 4.1** Make a bar chart of {1, 1, 2, 3, 5}.
- 4.2** Make a pie chart of numbers from 1 to 10.
- 4.3** Make a bar chart of numbers counting down from 20 to 1.
- 4.4** Display numbers from 1 to 5 in a column.
- 4.5** Make a number line plot of the squares {1, 4, 9, 16, 25}.

**4.6** Make a pie chart with 10 identical segments, each of size 1 (CostantArray).

**4.7** Make a column of pie charts, respectively with 1, 2 and 3 identical segments (CostantArray).

**+4.1** Make a **list** of pie charts with 1, 2 and 3 identical segments.

**+4.2** Make a bar chart of the sequence 1, 2, 3, ..., 9, 10, 9, 8, 7, ..., 1.

**+4.3** Make a **list** of a pie chart, bar chart and line plot of the numbers from 1 to 10.

**+4.4** Make a **list** of a pie chart and a bar chart of {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}.

**+4.5** Make a column of two number line plots of {1, 2, 3, 4, 5} .

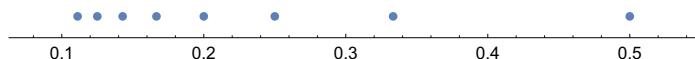
**+4.6** Make a number line of fractions  $\frac{1}{2}, \frac{1}{3}, \dots$  through  $\frac{1}{9}$  .

```

fractions = Range[2, 9]^-1;
fractions0 = 1 / Range[2, 9]
fractions == fractions0
NumberLinePlot[fractions]
{1/2, 1/3, 1/4, 1/5, 1/6, 1/7, 1/8, 1/9}

```

True



```

(* SetAttributes[....] *)
{Attributes[Power], Attributes[Divide]} // TableForm
Listable      NumericFunction      OneIdentity      Protected
Listable      NumericFunction      Protected

```

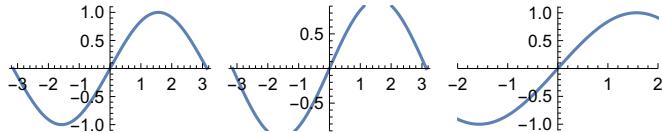
## Q & A

How is the vertical scale determined on plots?

It is set up to automatically include all points, except distant outliers.

The **PlotRange** option lets you specify the exact range of the plot.

```
p0 = Plot[ Sin[x], {x, -Pi, Pi}];  
p1 = Plot[ Sin[x], {x, -Pi, Pi}, PlotRange → {-0.9, 0.9}];  
p2 = Plot[ Sin[x], {x, -Pi, Pi}, PlotRange → {{-2, 2}, All}];  
GraphicsRow[{p0, p1, p2}, ImageSize → Medium]
```



### SetDelayed

Come detto all'inizio di questa sezione 2.2, in analogia alla chiamata di funzione in altri linguaggi, chiamiamo

**valore di ritorno** (return value) di una data espressione il risultato della valutazione di tale espressione.

In altre parole, diciamo che ogni espressione restituisce un'altra espressione come suo valore.

⌘ Esistono dei casi, pero', in cui l'affermazione precedente sembra essere falsa.

Un esempio e' dato dall'operatore SetDelayed (cfr. Sezione 2.3.4), il quale pare non restituire alcun valore:

```
s2 = Sqrt[2]
s3 := Sqrt[3]
(*     :=   e' il simbolo sintattico di SetDelayed  *)

 $\sqrt{2}$ 
```

### Null

L'esempio qui sopra pare implicare che non esista alcun valore di ritorno della SetDelayed.

⌘ In effetti, SetDelayed restituisce il simbolo speciale Null, che di norma non appare nell'output.

```
? Null
```

Null is a symbol used to indicate the absence of an expression or a result. It is not displayed in ordinary output. When Null appears as a complete output expression, no output is printed. ➤

```
expr1; expr2; expr3;
(* la valutazione qui sopra restituisce Null, che non viene mostrato *)
```

⌘ Null appare se e' parte di un'espressione piu' grande:

```
s2 = Sqrt[2];
(* Notiamo che serve il punto-e-virgola per inibire l'output di Set *)
1 + %

 $1 + \sqrt{2}$ 
```

```
s3 := Sqrt[3]
(* Notiamo che non serve il punto-e-
virgola per inibire l'output di SetDelayed *)
1 + %

1 + Null
```

⌘ DISGRESSIONE(su Null e sul punto-e-virgola)

```
(* Disgressione su Null e sul punto-e-virgola *)
f1[x_] := Module[{tmp = x},
  While[tmp > 2, tmp = Sqrt[tmp]];
  (* NOTIAMO il punto-e-virgola dopo While[] *)
  tmp
];
f1[100.]
```

```
f2[x_] := Module[{tmp = x},
  While[tmp > 2, tmp = Sqrt[tmp]]
  (* Nella Module,
  lo spazio tra While[] e statement tmp e' interpretato come prodotto *)*
  tmp
  (* Pertanto,
  viene restituito Null (esito di While[]) per il valore salvato in tmp *)
];
f2[100.]
```

1.77828

1.77828 Null

⌘ Null appare se sopprimiamo esplicitamente un output (magari perche' e' troppo grande, oppure perche' non ci interessa vederlo):

```
Timing[Total[Range[123 456]]]
(* Se siamo interessati solo al tempo di esecuzione,
sopprimiamo il risultato del calcolo *)
Timing[Total[Range[123 456]];]
```

{0.000402, 7 620 753 696}

{0.000097, Null}

⌘ SetDelayed[] e' un esempio di funzione che opera producendo un **effetto collaterale** (side effect): il risultato atteso dalla esecuzione della funzione non e' il **return value**, ma e' piuttosto un cambiamento apportato allo stato della sessione di **Mathematica** (oppure, in generale, al computer — ad esempio, la scrittura di dati in un file).

⌘ Nell'esempio visto, l'**effetto collaterale** e' la creazione di una regola di riscrittura per il simbolo s3

```
s3 := Sqrt[3]; s3^2
```

```
3
```

### Attenzione alle dipendenze cicliche

- ⌘ Per completare questa sezione, dobbiamo analizzare un ultimo argomento.
- Una assunzione implicita, nel processo di valutazione, e' che il sistema sia disegnato in modo che l'insieme di tutte le espressioni sia parzialmente ordinato rispetto alla valutazione stessa.
- In termini equivalenti, si suppone che (nel processo di valutazione) non esistano dipendenze cicliche.
- ⌘ Costruiamo un esempio in cui l'assunzione qui sopra venga violata:

```
(* Ricetta per un disastro! *)
yin := yang
(* il primo statement dice al Kernel che yin puo' essere riscritto come yang *)
yang := yin
(* il secondo statement dice al Kernel
   che yang puo' essere riscritto come yin *)
```

- ⌘ Per superare un caso come quello qui sopra, nel Kernel (per fortuna) e' built-in un interruttore di circuito (circuit breaker), detto *iteration limit*

```
yin
```

... \$IterationLimit: Iteration limit of 4096 exceeded.

```
Hold[yin]
```

- ⌘ Dopo che la riscrittura yin/yang e' avvenuta per 4096 volte, il Kernel avvolge il risultato corrente in una Hold[ ] (che inibisce ulteriori valutazioni) e restituisce, appunto, Hold[risultato\_corrente].

NOTA. Il fatto che, in questo esempio, il risultato finale sia lo stesso dell'espressione originale (Hold[yin] se si valuta yin, Hold[yang] se si valuta yang) e' una circostanza fortuita, dovuta alle definizioni usate.

- ⌘ Possiamo esaminare in dettaglio il processo ciclico, usando Trace[ ].

Prima, pero', modifichiamo il limite di iterazione, per mantenere l'output maneggiabile.

```
$IterationLimit = 20
(* Modifichiamo il valore di iteration limit , assegnando un valore ≥ 20
alla variabile globale $IterationLimit *)
Trace[yin]
```

```
20
```

**\$IterationLimit**: Iteration limit of 20 exceeded.

```
{yin, yang, yin, yang, yin, yang, yin, yang, yin, yang, yin, yang, yin, yang,
yin, yang, yin, yang, yin, yang, yin, {Message[$IterationLimit::itlim, 20],
{$IterationLimit::itlim, Iteration limit of `1` exceeded.},
{MakeBoxes[... $IterationLimit: Iteration limit of 20 exceeded., StandardForm],
TemplateBox[{$IterationLimit, itlim,
"Iteration limit of \!\\(\!*RowBox[{\"20\"}]\\) exceeded.", 2, 980, 14,
19 214 702 541 824 498 207, Local}, MessageTemplate]], Null}, Hold[yin]}
```

⌘ Trace[] mostra che i simboli yin e yang seguono un gioco del tipo *riscrittura a ping-pong* .

#### Message[]

La funzione Message[] causa l'apparizione del messaggio di errore.

⌘ La funzione Message[] viene invocata direttamente dal Kernel e non e' parte dell'espressione originale (e neppure di qualsiasi delle sue forme intermedie).

Message[] puo' inoltre essere chiamata direttamente (dal programmatore, per associare messaggi di errore o warnings alle funzioni che lei/lui scrive).

#### ? Message

Symbol

Message[symbol::tag] prints the message *symbol::tag* unless it has been switched off.

Message[symbol::tag,  $e_1, e_2, \dots$ ] prints a message, inserting the values of the  $e_i$  as needed.

▼

#### NOTA.

Prima di andare avanti, e' opportuno resettare il valore di **\$IterationLimit**

```
$IterationLimit = 4096;
```

### ■ Esercizio 1 pg. 30

Non comprendere bene il meccanismo di valutazione (Evaluation process) puo' essere fonte di errori comuni.

Perche', per esempio, *Mathematica* non restituisce un risultato in alta precisione dalla computazione numerica che segue?

```
tre = N[Sqrt[3.], 90]
```

```
1.73205
```

Usando FullForm, ci ricordiamo che Sqrt[3.] viene valutato immediatamente (a numero in precisione macchina). Quindi la funzione N non puo', successivamente, ampliare a 90 la sua precisione.

```
(* mach3 = Sqrt[3.]; tre=N[mach3 , 90]; *)
```

```
tre // FullForm
```

```
tre // Precision
```

```
1.7320508075688772`
```

```
MachinePrecision
```

Il modo corretto per ottenere Sqrt[3] con 90 cifre di precisione e' il seguente (cfr. sezione 2.1.2):

```
(* exact3 = Sqrt[3]; tre=N[exact3 , 90]; *)
```

```
tre90 = N[Sqrt[3], 90]
```

```
tre90 // Precision
```

```
1.73205080756887729352744634150587236694280525381038062805580697945193301690880`.
```

```
003708114619
```

```
90.
```

## 2.3 Forme speciali di Input (non nec)

### 2.3.1 Operatori aritmetici (non nec)

### 2.3.2 Operatori relazionali e booleani

## 2. Elementi fondamentali del linguaggio

### ■ Riscrittura di termini

#### 2.1 Espressioni: espressioni normali (2.1.1)

#### 2.1 Espressioni: atomi (2.1.2)

#### 2.2 Valutazione di espressioni

#### 2.3 Forme speciali di Input (non nec)

##### 2.3.1 Operatori aritmetici (non nec)

##### 2.3.2 Operatori relazionali e booleani

### ■ And, Or, Xor

Gli operatori relazionali e booleani di *Mathematica* sono gli stessi di C.

? Or

? Xor

Symbol

i

$e_0 \parallel e_1 \parallel \dots$  is the logical OR function. It evaluates its arguments in order, giving

True immediately if any of them are True, and False if they are all False.

▼

Symbol

i

Xor[ $e_0, e_1, \dots$ ] is the logical XOR (exclusive OR) function. It

gives True if an odd number of the  $e_n$  are True, and the rest are False. It

gives False if an even number of the  $e_n$  are True, and the rest are False.

▼

? Equal  
? Unequal  
? LessEqual

Symbol

i

*lhs == rhs* returns True if *lhs* and *rhs* are identical.

▼

Symbol

i

*lhs != rhs* or *lhs ≠ rhs* returns False if *lhs* and *rhs* are identical.

▼

Symbol

i

*x <= y* or *x ≤ y* yields True if *x* is determined to be less than or equal to *y*.

*x<sub>0</sub> ≤ x<sub>1</sub> ≤ x<sub>2</sub>* yields True if the *x<sub>k</sub>* form a nondecreasing sequence.

▼

7 > 4 && 2 ≠ 3

(\* Is (7 Greater than 4) And (2 Unequal 3) ?? \*)

True

7 ≤ 4 || 2 == 3

(\* Is (7 LessEqual than 4) Or (2 Equal 3) ?? \*)

False

⌘ Gli operatori booleani *And* e *Or* sono n-ari.

⌘ Come in C, gli operatori booleani “cortocircuitano” (vanno in short-circuit) una volta che il loro risultato e’ stato determinato.

Per esempio, nella espressione che segue, il primo argomento di *And* viene valutato come False, pertanto il secondo argomento non viene mai valutato (in questo modo, si evita di arrivare a dover valutare 1/0, che genererebbe un messaggio di errore):

```
1 < 0 && 1 / 0
```

False

⌘ Non tutte le parti di And sono valutate prima della head stessa (idem per Or).

In altre parole, per And ed Or viene usata la valutazione non-standard:

la head viene valutata per prima, mentre gli argomenti sono valutati sotto il controllo della funzione stessa (non prima che la funzione/head venga chiamata).

Verifichiamolo usando Attributes[]

```
Attributes[And]
```

```
Attributes[Or]
```

```
(* And ed Or hanno l'attributo HoldAll *)
```

```
{Flat, HoldAll, OneIdentity, Protected}
```

```
{Flat, HoldAll, OneIdentity, Protected}
```

? HoldAll

Symbol



HoldAll is an attribute that specifies that all arguments

to a function are to be maintained in an unevaluated form.



⌘ DIGRESSIONE (sugli Attributes)

? NHoldFirst

```
(* Prevents N[ ] from affecting the 1st argument of a function *)
```

```
SetAttributes[f, NHoldFirst]; {N[f[Pi, E, GoldenRatio]], N[g[Pi, E, GoldenRatio]]}
```

Symbol



NHoldFirst is an attribute which specifies that

the first argument to a function should not be affected by N.



```
{f[π, 2.71828, 1.61803], g[3.14159, 2.71828, 1.61803]}
```

(\* proprietà di idempotenza \*) ? **OneIdentity**

Symbol

*i*

OneIdentity is an attribute that can be assigned to a symbol  $f$  to indicate that  $f[x]$ ,  $f[f[x]]$ , etc. are all equivalent to  $x$  for the purpose of pattern matching.

(\* proprietà associativa \*) ? **Flat**

Symbol

*i*

Flat is an attribute that can be assigned to a symbol  $f$  to indicate that all expressions involving nested functions  $f$  should be flattened out. This property is accounted for in pattern matching.

(\* proprietà commutativa \*) ? **Orderless**

Symbol

*i*

Orderless is an attribute that can be assigned to a symbol  $f$  to indicate that the elements  $e_n$  in expressions of the form  $f[e_0, e_1, \dots]$  should automatically be sorted into canonical order. This property is accounted for in pattern matching.

? **Listable**

Symbol

*i*

Listable is an attribute that can be assigned to a symbol  $f$  to indicate that the function  $f$  should automatically be threaded over lists that appear as its arguments.



**? Protected**

Symbol i

Protected is an attribute that prevents  
any values associated with a symbol from being modified.

⌘ Esistono alcune differenze nell'uso degli operatori relazionali, rispetto al modo in cui essi vengono usati in C.

1. Possiamo usarli "in catena":

```
5 > 4 > 3
5 > 4 && 4 > 3
(5 > 4 > 3) == (5 > 4 && 4 > 3)
```

True

True

True

2. Non e' obbligatorio che gli argomenti di un operatore relazionale siano numeri.

Ovviamente, se gli argomenti non sono numerici, lo statement potrebbe rimanere non valutato (Unevaluated).

```
a == b
(* E' lecito scrivere uno statement quale quello qui sopra,
anche se l'esito di Equal, qui, e' non-valutato (Unevaluated) *)
```

a == b

⌘ Per ottenere sempre una Evaluation, si puo' usare SameQ[]

```
a === b
```

(\* dato che "a" e "b" sono manifestamente diversi, SameQ restituisce False \*)

(\* SameQ: numeri Real , diversi nell'ultimo bit, sono considerati identici \*)

```
? SameQ
```

```
False
```

Symbol

*i*

*lhs* === *rhs* yields True if the expression *lhs* is identical to *rhs*, and yields False otherwise.

▼

⌘ Mathematica conosce le proprieta' dell'addizione

```
a + b == b + a
```

```
a + b === b + a
```

```
True
```

```
True
```

⌘ La negazione logica di SameQ[] e' Unequal[]:

```
a != b
```

```
True
```

⌘ Come gia' detto, SameQ[] ed Unequal[] vengono *sempre* valutati e restituiscono True oppure False.

Questo non vale per Equal[] e per Unequal[], che possono rimanere non valutati.

```
{a == b, a === b, a ≠ b, a != b}
```

```
{a == b, False, a ≠ b, True}
```

### 2.3.3 Riutilizzazione di risultati (non nec)

### 2.3.4 Statement di assegnazione (non nec)

### 2.3.11 Trappole sintattiche per l'inconsapevole

Ci sono alcune caratteristiche del parser di *Mathematica* che sono intese a rendere l'input piu' naturale per un utente, ma che possono causare conseguenze non volute se non si sta attenti.

⌘ Prima di tutto, uno "spazio" puo' essere usato al posto dell'asterisco \* per significare la moltiplicazione. Questo ricorda molto la notazione matematica standard, in cui i simboli vengono moltiplicati tra loro

```
a b
(* a   spazio   b *)
```

```
a b
```

⌘ In alcuni casi speciali, lo spazio non e' necessario.

1. *Il primo caso e' quello in cui un numero e' seguito da un carattere non-numerico.*

*Dato che i nomi di simboli non possono iniziare con una cifra numerica (digit), "2a" e' inteso come moltiplicazione 2\*a*

```
2 a
(* 2a senza spazio in mezzo *)
```

2. *Il secondo caso (in cui lo spazio non e' necessario) e' quello in cui due simboli sono separati da un delimitatore, quale una parentesi tonda o una parentesi graffa (per una lista).*

*Questo e' vero per parentesi tonde o graffe, aperte (i.e. da sinistra) o chiuse (i.e. da destra).*

```
a (b + c)
(b + c) a
```

```
a (b + c)
```

```
a (b + c)
```

```
(* prodotto vettoriale di "a" per ogni elemento di una lista;
in questo caso, la lista ha un solo elemento "b+c" ; l'output e' una lista *)
a {b + c}
```

```
{b + c} a
```

```
{a (b + c)}
```

```
{a (b + c)}
```

```
a {b + c} // ExpandAll
{b + c} a // ExpandAll
```

 $\{a b + a c\}$ 
 $\{a b + a c\}$ 

**Nota.** Nel caso di parentesi quadra (singola o doppia), “a [b]” e’ una chiamata di funzione, mentre “a [[b]]” e’ una operazione di indicizzazione (subscripting operation).

Non sono invece lecite le scritture:  $[b+c]a$  oppure  $[[b]]a$ .

```
a[b + c]
a[[0]]
```

 $a[b + c]$ 

Symbol

 $[b + c] a$ 

... Syntax: Expression cannot begin with "[b + c] a".

 $[[b]] a$ 

... Syntax: Expression cannot begin with "[[b]] a".

**# Mathematica** mostra sempre, nell'output, uno spazio tra simboli che sono considerati distinti (anche se, nell'input, noi non abbiamo messo tale spazio). Viceversa, se nell'input non mettiamo uno spazio (i.e. “ab”), **Mathematica** non considera “a” e “b” come simboli distinti.

```
a (b) (* scriviamo questo input senza inserire spazi.
Nonostante cio', nell'output Mathematica mostra
" a spazio b" dato che "a" e "b" sono due simboli distinti*)
```

```
ab (* questo e' un solo simbolo *)
```

 $a b$ 
 $ab$ 

Vediamolo con un esempio.

$$\{a b / a, \ ab / a\}$$

(\* Il primo elemento della lista e'  $a*b/a$ , che puo' essere semplificato.  
Il secondo elemento della lista e'  $ab/a$ , che non puo' essere semplificato.\*)

$$\left\{ b, \frac{ab}{a} \right\}$$

⌘ Dato che l'uso e la interpretazione di uno spazio possono causare errori, e' buona regola usare sempre gli spazi (ed eventualmente anche le parentesi), anche quando non e' strettamente necessario: questo aiuta ad evitare errori e puo' migliorare la leggibilita' del codice.

⌘ La seconda caratteristica che puo' trarre in inganno l'utente e' specifica del notebook di interfaccia ed e' collegato alla possibilita' di scrivere in input molteplici espressioni in una sola cella (di input, appunto).

⌘ Data tale possibilita', il problema nasce quando vogliamo dare in input una espressione che non sta su un'unica linea di input: Mathematica potrebbe tentare di interpretare tale espressione come espressione multipla.

```
3 * 4 + 5
```

```
* 2
```

(\* Questa espressione, che dovrebbe essere unica e stare su una sola riga,  
viene interpretata come 2 espressioni distinte  
(perche' la seconda riga e' su una newline).

La prima 3\*4+5 restituisce 17.

La seconda \*2 causa un errore nel parser \*)

17

⌘ Una situazione peggiore e' quella in cui la singola espressione viene interpretata come espressione multipla, su piu' righe, e non intervengono errori di sintassi, per cui l'utente non riceve alcun messaggio di errore

```
3 * 4
```

```
+ 5 * 2
```

(\* Questa espressione, che dovrebbe essere unica e stare su una sola riga, viene interpretata come 2 espressioni distinte.

La prima  $3*4$  restituisce 12.

Nella seconda  $+5*2$ ,  $+5$  viene interpretato come operatore unario, per cui complessivamente il risultato e' 10 \*)

12

10

⌘ Per evitare questo problema, bisogna essere sicuri di non terminare/interrompere mai una linea intermedia di input in un modo che permetta a Mathematica di reinterpretare tale linea come espressione completa (a meno che non lo sia davvero, ovviamente).

```
3 * 4 +
```

```
5 * 2
```

(\* Questa espressione dovrebbe essere unica e stare su una sola riga: interrompendola col "+" finale, essa non puo' venire interpretata come 2 espressioni distinte: Mathematica sa che ci deve essere altro input dopo il "+" e lo va a prendere sulla seconda riga.

Questo e' un modo, di eseguire l'esempio, che porta ad un risultato corretto \*)

22

⌘ Se ci sono delimitatori disaccoppiati (e.g. parentesi aperte, tonde, quadre o graffe, non chiuse alla fine di una linea), Mathematica non considera tale espressione come completa, per cui prosegue sulla linea successiva, fino a trovare il delimitatore (di chiusura) corrispondente.

```
Sqrt[3 * 4  
+ 5 * 2]
```

$\sqrt{22}$

⌘ Se si valuta una cella il cui input termina con un delimitatore aperto, si riceve ovviamente un messaggio di errore di sintassi.

```
Sqrt[ 3 * 4
```

⌘

■ **Esercizio 1 pg. 46**

⌘ In quale modo Mathematica interpreta ciascuna delle espressioni che seguono? Perche'?

```
a^bc
a^2 bc
(* bc e' un unico simbolo *)
```

```
abc
```

```
a2 bc
```

```
a^bc // FullForm
a^2 bc // FullForm
(* L'esponenziale ha precedenza sulla moltiplicazione,
per cui a^2 bc e' interpretato come (a^2)*bc *)
```

```
Power[a, bc]
```

```
Times[Power[a, 2], bc]
```

■ **Esercizio 2 pg. 46**

⌘ Quale e' il risultato della valutazione, in una singola cella di input, degli input che seguono? E' chiaro quello che succede?

```
{
  2+3,
  4+5 *
    6+7
}
(* la prima componente della lista e' 2+3 che viene interpretato come 5 *)
(* la seconda componente della lista e' diviso su due righe:
  4+5 e 6+7, che vengono interpretate come una unica espressione 4+5*6+7 *)
```

```
{5, 41}
```

**2.3.12 Informazioni sui simboli (non nec)**

# Operations on Lists

There are thousands of **built-in** functions to work with lists.

You can do arithmetics with lists :

```
{1, 2, 3} + 10
Plus[{1, 2, 3}, 10];
{1, 2, 3} + {1, 1, 2}
Plus[{1, 2, 3}, {1, 1, 2}];
{11, 12, 13}

{2, 3, 5}

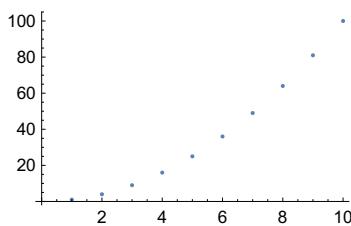
3 {1, 2, 3}
Times[3, {1, 2, 3}];
{1, 1, 2} * {1, 2, 3}
Times[{1, 1, 2}, {1, 2, 3}];
{3, 6, 9}

{1, 2, 6}

(* A . B *)
Dot[{1, 1, 2}, {1, 2, 3}];
{1, 1, 2} . {1, 2, 3}
9
```

## Range and ListPlot

```
myRange = Range[10]^2
ListPlot[myRange, ImageSize → Small]
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```



## Sort a list into order

```
Sort[{4, 2, 1, 3, 6}]
```

```
{1, 2, 3, 4, 6}
```

Look at the rules followed by **Sort** at its Help Page.

**FullForm** prints the full form of each expression in the list (it helps to understand the output of Sort).

```
(* Sort orders integers, rational,
approximate real numbers by their numerical values *)
(* Sort orders complex numbers by their real parts and,
in a tie, by the absolute values of their imaginary parts *)
(* Sort orders symbols by their names *)
(* Sort orders strings as in a dictionary, with lowercase before uppercase. *)
(* Mathematical operators are from-higher-to-lower precedence. *)
testList = {1, z, a, 1/2, π, Sqrt[3], (* 3^(1/2), *) 0.7, 0, E};
mysort = Sort[testList]
FullForm[mysort]
```

$$\left\{0, \frac{1}{2}, 0.7, 1, \sqrt{3}, a, e, \pi, z\right\}$$

```
List[0, Rational[1, 2], 0.7` , 1, Power[3, Rational[1, 2]], a, E, Pi, z]
```

```
Sort[{1 + 2 I, 1 + I}]
```

```
FullForm[%]
```

$$\{1+i, 1+2i\}$$

```
List[Complex[1, 1], Complex[1, 2]]
```

**Note on Imaginary Unit  $I$  and Nepero number  $E$**

```
(* Non posso chiedere questa Set *)
```

```
I = 2
```

**Set:** Symbol  $i$  is Protected.

```
2
```

```
(* E, I, D[], N[] *)
```

```
{Exp[x], Exp[1]}
```

$$\{e^x, e\}$$

## Length finds how long a list is

```
vec = {5, 3, 4, 5, 3, 4, 5};
vlen = Length[vec];
vdim = Dimensions[vec];
Print["vector ", vec, " of len=", vlen, " and dims=", vdim];
MatrixForm[vec]

vector {5, 3, 4, 5, 3, 4, 5} of len=7 and dims={7}
```

```


$$\begin{pmatrix} 5 \\ 3 \\ 4 \\ 5 \\ 3 \\ 4 \\ 5 \end{pmatrix}$$


(*1x7 matrix*)
row = {vec}
rowlen = Length[{vec}];
rowdim = Dimensions[{vec}];
Print["row-matrix ", row, " of dims=", rowdim, " and len=", rowlen];
MatrixForm[row]
{{5, 3, 4, 5, 3, 4, 5}}
row-matrix {{5, 3, 4, 5, 3, 4, 5}} of dims={1, 7} and len=1
(5 3 4 5 3 4 5)

(* 2x3 matrix *)
mat = {{1, 2, 3}, {4, 5, 6}};
matlen = Length[mat];
matdim = Dimensions[mat];
Print["matrix ", mat, " of dims=", matdim, " and len=", matlen];
(* a##&     equivalente a definire a__ come argomento di funzione *)
(* a__     indica uno o piu' argomenti *)
{MatrixForm[mat],
 MatrixForm[Array[a## &, {2, 3}]]}
matrix {{1, 2, 3}, {4, 5, 6}} of dims={2, 3} and len=2
{{1 2 3}, {4 5 6}}, {{a_{1,1} a_{1,2} a_{1,3}}, {a_{2,1} a_{2,2} a_{2,3}}}

(* list of lists *)
tab = {{4, 5}, {6, 7, 8, 9}};
tablen = Length[tab];
tabdim = Dimensions[tab];
Print["list of lists ", tab, " of dims=", tabdim, " and len=", tablen];
TableForm[tab]
list of lists {{4, 5}, {6, 7, 8, 9}} of dims={2} and len=2
4      5
6      7      8      9

```

```
(* 2x3x2 tensor *)
tens = Table[i + j * k, {i, 2}, {j, 3}, {k, 2}];
tenslen = Length[tens];
tensdim = Dimensions[tens];
Print["tensor ", tens, " of dims=", tensdim, " and len=", tenslen];
(*{MatrixForm[Transpose[tens[[1]]]],
 MatrixForm[Transpose[tens[[2]]]]}*)
(* ## stands for one or more arguments *)
{MatrixForm[tens],
 MatrixForm[Array[a## &, {2, 3, 2}]]}
tensor {{2, 3}, {3, 5}, {4, 7}}, {{3, 4}, {4, 6}, {5, 8}} of dims={2, 3, 2} and len=2

$$\left\{ \begin{pmatrix} (2) \\ (3) \end{pmatrix} \begin{pmatrix} (3) \\ (5) \end{pmatrix} \begin{pmatrix} (4) \\ (7) \end{pmatrix}, \begin{pmatrix} (3) \\ (4) \end{pmatrix} \begin{pmatrix} (4) \\ (6) \end{pmatrix} \begin{pmatrix} (5) \\ (8) \end{pmatrix} \right\}, \left\{ \begin{pmatrix} a_{1,1,1} \\ a_{1,1,2} \end{pmatrix} \begin{pmatrix} a_{1,2,1} \\ a_{1,2,2} \end{pmatrix} \begin{pmatrix} a_{1,3,1} \\ a_{1,3,2} \end{pmatrix}, \begin{pmatrix} a_{2,1,1} \\ a_{2,1,2} \end{pmatrix} \begin{pmatrix} a_{2,2,1} \\ a_{2,2,2} \end{pmatrix} \begin{pmatrix} a_{2,3,1} \\ a_{2,3,2} \end{pmatrix} \right\}$$

```

**Total** gives the total from adding up a list

```
Total[{1, 1, 2, 2}];
```

```
Total[Range[10]]
```

```
55
```

**Count** the number of times something appears in a list

```
Clear[d];
Count[{a, b, a, a, c, b, a}, a]
Count[{a, b, a, a, c, b, a}, d]
Count[{a, b, a, 1, c, b, a}, d]
d = 1;
Count[{a, b, a, 1, c, b, a}, d]
4
0
0
1

(* Clear, ClearAll, ClearAttributes *)
Clear[a, b, c, d];
a = 1;
? a
```

Symbol
Global`a
Assignment
a = 1
Full Name Global`a
^

```
Clear["Global`*"]
```

```
? a
```

Symbol
Global`a
Full Name Global`a
^

## Use **First**, **Last**, **Part**, **Rest**, **Most**, to get elements of a list

```
alist = {7, 9, 4, 3, 1, 0, 5};
{First[alist], Last[alist], Part[alist, 2]}
Part[alist, 2] == alist[[ 2 ]]
```

```
{7, 5, 9}
```

```
True
```

Picking out the first element in a sorted list is equivalent to finding its minimum element:

```
blist = {6, 7, 1, 2, 4, 5};
First[ Sort[blist] ] == Min[blist]
```

```
True
```

**Rest** gives all the elements in a list after the first one.

**Most** gives all elements in a list except the last one.

```
blist = {6, 7, 1, 2, 4, 5}
blist[[0]]
(* Drop = scartare, buttare *)
{Rest[blist], Rest[blist] == Drop[blist, 1]}
{Most[blist], Most[blist] == Drop[blist, -1]}
{6, 7, 1, 2, 4, 5}
List
{{7, 1, 2, 4, 5}, True}
```

```
{6, 7, 1, 2, 4}, True}
```

## **IntegerDigits** makes a list of the digits in an Integer

The default is "decimal" digits:

```
IntegerDigits[203]
{2, 0, 3}

(* 0203 == 203 *)
IntegerDigits[203] == IntegerDigits[0203]
True

(* this returns unevaluated *)
IntegerDigits[o203]
IntegerDigits[o203]
```

You can specify any base, e.g., "binary" :

```
IntegerDigits[203, 2]
{1, 1, 0, 0, 1, 0, 1, 1}

IntegerDigits[203, 16]
{12, 11}
```

**FromDigits** reconstructs an Integer from its list of digits:

```
FromDigits[{2, 0, 3}]
203

FromDigits[{2, 0, 3}] == FromDigits[{0, 2, 0, 3}]
True

FromDigits[{1, 1, 0, 0, 1, 0, 1, 1}, 2]
203
```

Digits larger than the base are “carried”

```
FromDigits[{2, 1, 0, 0, 1, 0, 1, 1}, 2] == FromDigits[{1, 0, 1, 0, 0, 1, 0, 1, 1}, 2]
True

FromDigits[{3, 1, 0, 0, 1, 0, 1, 1}, 2] == FromDigits[{1, 1, 1, 0, 0, 1, 0, 1, 1}, 2]
True
```

FromDigits is the inverse of IntegerDigits.

Since IntegerDigits discards the sign, FromDigits[IntegerDigits[n]]==Abs[n]:

```
FromDigits[IntegerDigits[10]] == Abs[10]
True
```

## Take or Drop a specified number of elements from a list

Look at the Help Pages of **Take** and **Drop**

```
mylist = {11, 23, 41, 0, 62, 32, 12};
Take[mylist, 3]
{11, 23, 41}

Drop[mylist, 3]
{0, 62, 32, 12}
```

## Vocabulary

{2,3,4}+{5,6,2}	arithmetics on lists
<b>Sort</b> [{5,7,1}]	sort a list into order
<b>Length</b> [{3,3}]	length of a list (number of elements)
Dimensions	
<b>Total</b> [{1,1,2}]	total of all elements in a list
<b>Count</b> [{3,2,3},3]	count occurrences of an element
First[{2,3}]	first element in a list
Last[{6,7,8}]	last element in a list
<b>Part</b> [{3,1,4},2]	part of a list, also written as {3, 1, 4}[[2]]
<b>Take</b> [{6,4,3,1},2]	take elements from a list
Drop[{6,4,3,1},2]	drop elements from a list
<b>IntegerDigits</b> [1234]	list of digits in an integer
<b>FromDigits</b> [{1,2,3,4}]	an integer from its digits
Rest[{6,4,3,1}]	all the elements of a list after the first one
Most[{6,4,3,1}]	all elements of a list except the last one
Dot	
MatrixForm	
Clear, ClearAll, ClearAttributes	
Map	

## Exercises

**5.1** Make a list of the first 10 squares, in reverse order .

**5.2** Find the Total of the first 10 squares.

**5.3** Make a plot of the first 10 squares, starting at 1.

**5.4** Use Sort, Join, Range, to create {1, 1, 2, 2, 3, 3, 4, 4}.

**5.5** Use Range and Plus to make a list of numbers from 10 to 20, inclusive.

```
Clear[k];
Range[k, k + 10]
(* Range[imin, imax] generates {imin, ..., imax} *)
sol = Range[k, k + 10] + (10 - k)
{k, 1 + k, 2 + k, 3 + k, 4 + k, 5 + k, 6 + k, 7 + k, 8 + k, 9 + k, 10 + k}
{10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}

sol == Range[10, 20] + 0 == Range[9, 19] + 1 ==
(* ... == *) Range[5, 15] + 5 == Range[1, 11] + 9 == Range[0, 10] + 10
True

Clear[data];
data[k_] := Range[k, k + 10] + (10 - k)
sol == data[10] == data[9] == data[5] == data[1] == data[0] == data[321]
True
```

**5.6** Make a combined list of the first 5 squares and cubes, sorted into order.

**5.7** Find the number of digits in  $2^{128}$ .

**5.8** Find the first digit of  $2^{32}$ .

**5.9** Find the first 10 digits in  $2^{100}$ .

**5.10** Find the largest digit that appears in  $2^{20}$ .

**5.11** Find how many zeros appear in the digits of  $2^{1000}$ .

**5.12** Use Part, Sort, IntegerDigits, to find the 2nd-smallest digit in  $2^{20}$ .

```
Clear[test, itest, sid];
test = 2^20;
itest = IntegerDigits[test];
sid = Sort[itest];
(* Usare Union[sid][[2]] per rimuovere di eventuali duplicati/molteplicita' *)
(* Qui non serve, perche' test=2^10=1048576 non ha cifre duplicate *)
{test, Part[sid, 2]}
(* Part[sid,2] == sid[[2]] *)
{1048576, 1}
```

**5.13** Make a line plot of the sequence of digits that appear in  $2^{128}$ .

**5.14** Use **Take** and **Drop** to get the sequence 11 through 20 from **Range[100]**.

```
Clear[data, data11to100, sol1, data1to20, sol2, td, sol4];
(* interi da 1 a 100*)
data = Range[100];

(* modo 1 : elimino i primi dieci numeri in data;
poi prendo i primi dieci numeri di data11to100 *)
data11to100 = Drop[data, 10];
sol1 = Take[data11to100, 10]
{11, 12, 13, 14, 15, 16, 17, 18, 19, 20}

(* modo 2 : prendo i primi venti numeri in data;
poi elimino i primi dieci numeri di data1to20 *)
data1to20 = Take[data, 20];
sol2 = Drop[data1to20, 10];
sol1 == sol2
True

(* modo 3 : Part *)
sol1 == data[[11 ;; 20]]
True
```

```
(* modo 4 : TakeDrop[list, {n,m}] restituisce {list1, list2}, in cui
   list1 e' fatta dagli elementi da n ad m di list,
   list2 e' fatta dai restanti elementi di list *)
td = TakeDrop[data, {11, 20}];
sol4 = First[td];
sol1 == sol4
True
```

**+5.1** Make a list of the first 10 multiples of 3 (0 excluded).

**+5.2** Make a list of the first 10 squares using **Range** and **Times** and no other built-in.

**+5.3** Find the last digit of  $2^{37}$ .

**+5.4** Find the penultimate digit of  $2^{32}$ .

```
Clear[test, itest, itestMeno1, penultimate];
test = 2^32;
itest = IntegerDigits[test];

(* modo 1 *)
(* Part[itest,-2]==itest[-2] *)
penultimate = Part[itest, -2]

(* modo 2 *)
penultimate == Part[itest, Length[itest] - 1]

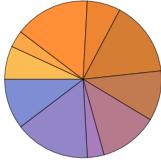
(* modo 3: Most returns itest with the Last element removed *)
itestMeno1 = Most[itest];
penultimate == Last[itestMeno1]

(* modo 4 : Take[expr, {n}] restituisce solo l'elemento n-esimo,
pero' in forma di singoletto *)
singoletto = Take[itest, {-2}];
penultimate == First[singoletto]
9
True
True
True
```

**+5.5** Find the sum of all the digits of  $3^{126}$ .

**+5.6** Make a PieChart of the sequence of digits in  $2^{32}$ .

```
Clear[test, data];
test = 2^32;
data = IntegerDigits[test];
PieChart[data, ImageSize → Tiny];
(* lhs := rhs *)
(* nomeFun[var_] := body *)
(* := SetDelayed *)
(* = Set *)
(* NO: pcid0[ t_ ]=PieChart[IntegerDigits[t],ImageSize→Tiny]; *)
pcid[t_] := PieChart[IntegerDigits[t], ImageSize → Tiny];
pcid[test]
```

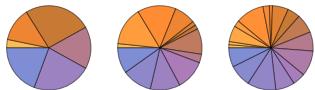


**+5.7** Make a list of pie charts for the sequence of digits in  $2^{20}, 2^{40}, 2^{60}$ .

```
(* modo1: OK, ma non e' bella programmazione *)
GraphicsRow[
{PieChart[IntegerDigits[2^20]],
 PieChart[IntegerDigits[2^40]],
 PieChart[IntegerDigits[2^60]]}, ImageSize → Small];

(* modo2: meglio, ma migliorabile *)
Clear[pcid];
pcid[t_] := PieChart[IntegerDigits[t], ImageSize → Tiny];
GraphicsRow[{pcid[2^20],
 pcid[2^40],
 pcid[2^60]}}, ImageSize → Small];
```

```
(* modo3: Array e Map *)
(* Array[f,n] genera una lista {f[1], f[2], ..., f[n]} *)
(* Map[f, expr] applica f ad ogni elemento a livello 1 di expr *)
Clear[data, pcid];
pcid[t_] := PieChart[IntegerDigits[t], ImageSize → Tiny];
(* {2^20,2^40,2^60} == Array[2^(20 #)&, 3 ] == Table[2^(20 k),{k,3}] *)
data = Array[2^(20 #)&, 3];
GraphicsRow[ Map[pcid, data], ImageSize → Small]
```



## Q & A

Can one add lists of different lengths? NO

```
{1, 2} + {1, 2, 3};
{1, 2} * {1, 2, 3};
{1, 2} ^ {1, 2, 3};
```

... Thread: Objects of unequal length in {1, 2} + {1, 2, 3} cannot be combined.

... Thread: Objects of unequal length in {1, 2} {1, 2, 3} cannot be combined.

... Thread: Objects of unequal length in {1, 2}^{1,2,3} cannot be combined.

... Thread: Objects of unequal length in {0, Log[2]} {1, 2, 3} cannot be combined.

```
{1, 2, 0} + {1, 2, 3}
{1, 2, 0} * {1, 2, 3}
{1, 2, 0} ^ {1, 2, 3}

{2, 4, 3}
{1, 4, 0}
{1, 4, 0}
```

Can there be a list with nothing in it? YES, the empty list.

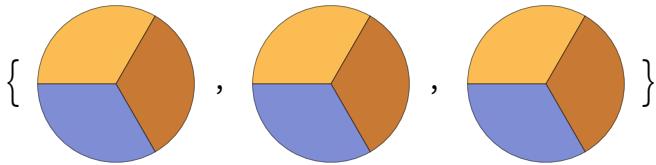
```
emptyList = {}  
Length[emptyList]  
Dimensions[emptyList]  
{}  
0  
{0}
```

# Making Tables

One of the most common and flexible ways to make lists is with **Table**.

In its simplest form, **Table** makes a list with a single element repeated a specified number of times.

```
Table[5, 10]
(* Table[5,10]==ConstantArray[5,10] *)
Table[x, 10]
Table[{1, 2}, 10]
Table[PieChart[{1, 1, 1}, ImageSize → Tiny], 3]
{5, 5, 5, 5, 5, 5, 5, 5, 5}
{x, x, x, x, x, x, x, x, x, x}
{{1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}, {1, 2}}
```



To make a **Table** where elements are not identical, we can introduce a variable, and iterate over it:

```
Table[a[n], {n, 5}]
(* Table[a[n],{n,5}]==Table[a[n],{n,1,5,1}]  *)
Table[n + 1, {n, 10}]
Table[n^2, {n, 10}]
{a[1], a[2], a[3], a[4], a[5]}
{2, 3, 4, 5, 6, 7, 8, 9, 10, 11}
{1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

Inside Table, the list `{n, 5}` collects the **variable** `n` and its **range** 5 (remember that a list is always a way of collecting things together): this kind of use of a list is called an *iterator specification*.

The iterator specification list, inside Table, allows to generalize to multidimensional arrays:

```

mat35 = Table[x^2 - y^2, {y, 3}, {x, 5}];
mat53 = Table[x^2 - y^2, {x, 5}, {y, 3}];
mat35 == Transpose[mat53]
{MatrixForm[mat35], MatrixForm[mat53]}

True

```

$$\left\{ \begin{pmatrix} 0 & 3 & 8 & 15 & 24 \\ -3 & 0 & 5 & 12 & 21 \\ -8 & -5 & 0 & 7 & 16 \end{pmatrix}, \begin{pmatrix} 0 & -3 & -8 \\ 3 & 0 & -5 \\ 8 & 5 & 0 \\ 15 & 12 & 7 \\ 24 & 21 & 16 \end{pmatrix} \right\}$$

You can make tables of anything.

```

Table[Range[n], {n, 5}]
Table[Column[Range[n]], {n, 8}]
{{1}, {1, 2}, {1, 2, 3}, {1, 2, 3, 4}, {1, 2, 3, 4, 5}}

```

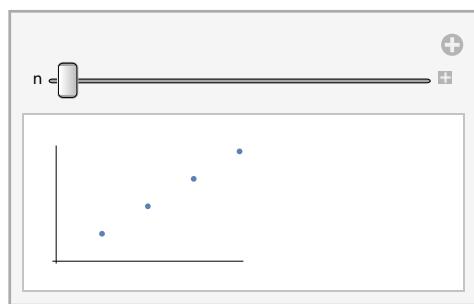
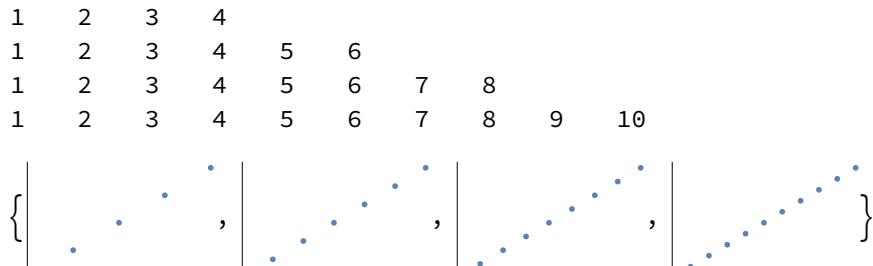
$$\left\{ 1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{6}, \frac{1}{7}, \frac{1}{8} \right\}$$

A table of plots, of longer and longer lists, produced by Range :

```
(* Creates Range[2*2] i.e. {1,...,2*2},
then Range[2*3] i.e. {1,...,2*3},
then Range[2*4] i.e. {1,...,2*4},
then Range[2*5] i.e. {1,...,2*5} *)
Table[
  Range[2 n],
  {n, 2, 5, 1}] // TableForm

Table[
  ListPlot[Range[2 n], PlotStyle -> PointSize[0.03], Ticks -> None, ImageSize -> Tiny],
  {n, 2, 5, 1}]

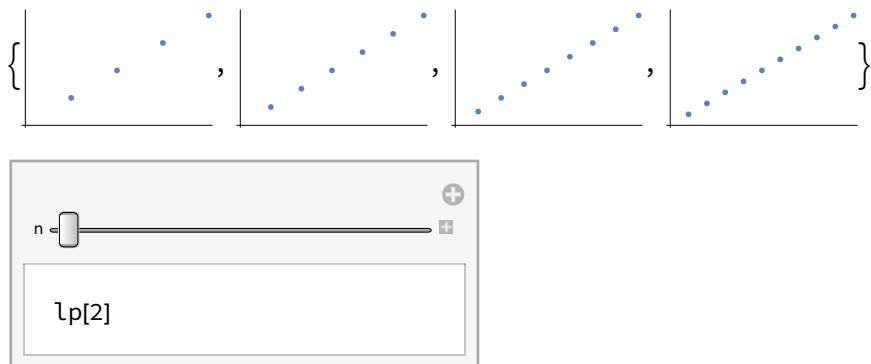
Manipulate[
  ListPlot[Range[2 n], PlotStyle -> PointSize[0.03], Ticks -> None, ImageSize -> Tiny],
  {n, 2, 5, 1}]
```



```
(* Define a function *)
lp[n_] :=
ListPlot[Range[2 n], PlotStyle -> PointSize[0.03], Ticks -> None, ImageSize -> Tiny]

Table[lp[n], {n, 2, 5, 1}]
```

```
Manipulate[lp[n], {n, 2, 5, 1}]
```

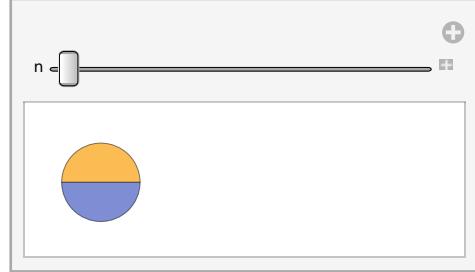
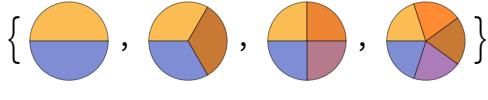


A table of pie charts with more and more slices :

```
data[n_] := Table[1, n]
```

```
Table[PieChart[data[n]], ImageSize -> 50], {n, 2, 5, 1}]
```

```
Manipulate[PieChart[data[n]], ImageSize -> 50], {n, 2, 5, 1}]
```



The iterator can take any name.

```
Table[2^expt, {expt, 10}]
{2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}
```



In *Mathematica*, consistency is fundamental.

Thus, e.g., **Range** is set up to deal with starting points and steps just like **Table**:

```
Range[10]
Range[4, 10]
Range[4, 10, 3]
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
{4, 5, 6, 7, 8, 9, 10}
{4, 7, 10}
```

The step does not need to be integer:

```
(* start+step, here: 0 + 0.1 *)
Table[n, {n, 0, 1, 0.1}]
Range[0, 1, 0.1]
{0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.}
{0., 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.}
```

**Table** and **Range** can deal with negative numbers (make sure to use the appropriate iterator specification):

```
Table[n, {n, -3, 2}] == Range[-3, 2] == {-3, -2, -1, 0, 1, 2}
```

```
Table[n, {n, 2, -3}] == Range[2, -3] == {}
```

```
Table[n, {n, 2, -3, -1}] == Range[2, -3, -1] == {2, 1, 0, -1, -2, -3}
```

```
Table[n, {n, -5, -3}] == Range[-5, -3] == {-5, -4, -3}
```

```
True
```

```
True
```

```
True
```

```
True
```

Range and Table go as far as the step take them, potentially stopping before the upper limit. E.g. Range[1, 6, 2] stops at 5 and gives {1, 3, 5}.

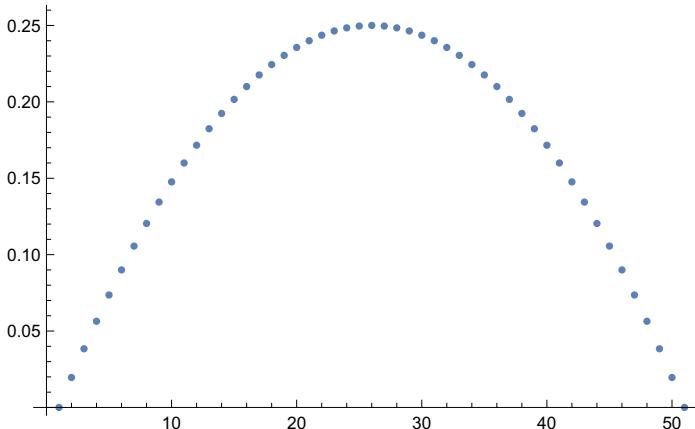
```
Table[n, {n, 1, 6, 2}] == Range[1, 6, 2] == {1, 3, 5}
```

## Different ways to perform the same task

```
mytab = Table[ x - x^2 , {x, 0, 1, .02} ];
lpt = ListPlot[ mytab ];
```

```
myrange = Range[0, 1, .02];
lpr = ListPlot[ myrange - myrange^2 ]
```

```
lpt == lpr
```



True

Table separately computes each entry in the list it generates (see it, using RandomInteger in Table).

This generates 20 independent random integers (with size up to 10) :

```
SeedRandom[3];
Table[ RandomInteger[10], 20]
{7, 10, 8, 2, 8, 0, 9, 10, 9, 1, 0, 2, 3, 9, 6, 0, 4, 5, 8, 6}
```

**RandomInteger** can generate the list directly (1! call):

```
SeedRandom[3];
RandomInteger[10, 20]
```

## Vocabulary

Table[ x, 5 ]	list of 5 copies of x
Table[ f[n] ,{ n, 10 } ]	list of values of f[n] with n from 1 (default) up to 10
Table[ f[n] ,{n, 2, 10 } ]	list of values with n from 2 to 10
Table[ f[n] ,{ n, 2, 10, 4} ]	list of values with n from 2 to 10 in steps of 4

Range[ 5 , 10 ]	list of numbers from 5 to 10
Range[ 10 , 20 , 2 ]	list of numbers from 10 to 20 in steps of 2
RandomInteger[ 10 , 20 ]	list of 20 random integers with size up to 10
Manipulate	
ReplaceAll	

---

## Exercises

- 6.1** Make a list in which the number 1000 is repeated 5 time.
- 6.2** Make a table of the values of  $n^3$  for  $n$  from 10 to 20.
- 6.3** Make a number line plot of the first 20 squares.
- 6.4** Make a list of the even number up to 20 (0 excluded).
- 6.5** Use Table to get the same result as Range[10].
- 6.6** Make a bar chart of the first 10 squares.
- 6.7** Make a table of the lists of (integer) digits forming each of the first 10 squares.
- 6.8** Make a list line plot of the length of the lists of (integer) digits in the first 100 squares.
- 6.9** Make a table of the first digit of the first 20 squares.
- 6.10** Make a list line plot of the first digits of the first 100 squares.
- +6.1** Make a list of the differences between  $n^3$  and  $n^2$  with  $n$  up to 10.
- +6.2** Make a list of the odd numbers up to 100.
- +6.3** Make a list of the squares of even numbers up to 100 (0 excluded).
- +6.4** Create the list  $\{-3, -2, -1, 0, 1, 2\}$  using Range.
- +6.5** Make a list for numbers  $n$  up to 12, in which each element is a column of the values of  $n$ ,  $n^2$  and  $n^3$ .
- +6.6** Make a list line plot of the last digits of the first 100 squares.
- +6.7** Make a list line plot of the first digit of the first 100 multiples of 3.
- +6.8** Make a list line plot of the total of the digits for each integer up to 200.

**+6.9** Make a list line plot of the total of the digits for each of the first 100 squares.

**+6.10** Make a number line plot of the numbers  $1/n$  with  $n$  from 1 to 20.

**+6.11** Make a line plot of a list of 100 random integers where the  $n$ -th integer is between 0 and  $n$ .

## Q & A

What are the constraints on the names of variables?

They can be any sequence of letters or numbers, but they cannot start with a number and (to avoid possible confusion with built-in functions) they should not start with a capital letter.

```
Variable = 6 ;(* DO NOT USE IT *)
myVariable = 2 ;(* suggested OK *)
variable2 = 4 ;(* OK *)
2 variable
FullForm[2 variable]
2 variable
Times[2, variable]

$MachinePrecision
15.9546

I
E
i
e
```

## Tech Notes

## 2.3.6 Definizione di funzione

*Mathematica* ci permette di definire nostre funzioni.

Qui sotto definiamo z come funzione dei due parametri x ed y:

```
z[x_, y_] := x + y
(* z[x_, y_] dichiarazione della funzione
   x+y corpo del programma *)
```

SetDelayed (forma speciale di input)  
:= → assegnazione differita  
z → nome di funzione

⌘ L'espressione alla sinistra (lhs) dell'assegnazione e' detta *dichiarazione* (declaration) della funzione.  
L'espressione alla destra (rhs) dell'assegnazione e' detta *corpo* (body) della funzione.

⌘ Ogni volta che la funzione viene usata in una espressione, si dice che tale funzione e' *chiamata* (called): il valore calcolato dalla funzione (return value) viene sostituito al posto della chiamata (function call).

```
z[Sqrt[3], 2 Pi]
(* quando valuto questa cella,
la funzione z che ho definito prima viene chiamata:
i valori Sqrt[3] e 2 Pi vengono sostituiti al posto di x ed y,
rispettivamente, ovunque x ed y siano presenti nel corpo della funzione *)
```

$$\sqrt{3} + 2\pi$$

⌘ Nell'esempio qui sopra, x ed y sono detti **parametri formali**, mentre i valori Sqrt[3] e 2 Pi sono detti **parametri attuali** oppure argomenti.

### Nota.

I valori dei parametri formali x ed y **non** dipendono dai valori dei simboli globali x ed y.  
Vediamolo con un esempio:

```
Clear[x, y]
x := 1; y := Log[2]; x + y
? x
? y
```

1 + Log[2]

Symbol
Global`x
Assignment
x := 1
Full Name Global`x
^

Symbol
Global`y
Assignment
y := Log[2]
Full Name Global`y
^

⌘ Dunque, x ed y sono ora variabili Globali con valori assegnati.

Nonostante cio' e nonostante le assegnazioni differite

x:=1;

y:=Log[2];

il valore di ritorno di z[ Sqrt[3], 2 Pi ] rimane invariato.

```
z[ $\text{Sqrt}[3]$ ,  $2 \pi$ ]
```

```
? z
```

```
 $\sqrt{3} + 2 \pi$ 
```

Symbol

Global`z

Definitions

$z[x_, y_] := x + y$

Full Name Global`z

^

⌘ Questo accade perche' abbiamo definito z con una SetDelayed ed essa rimane invariata nel contesto Global.

Se avessi definito z con una Set, allora il rhs sarebbe stato valutato immediatamente: di conseguenza, ci sarebbe stata una sostituzione immediata (nel corpo della funzione) di qualsiasi valore pre-esistente, dato ad x ed y.

### Ricapitolando.

```
Clear[z, x, y];
x := 1; y := Log[2];
(* Definisco z con SetDelayed *)
z[x_, y_] := x + y;
z[ $\text{Sqrt}[3]$ ,  $2 \pi$ ]
z[x, y]
```

```
 $\sqrt{3} + 2 \pi$ 
```

```
1 + Log[2]
```

```
Clear[z, x, y];
x := 1; y := Log[2];
(* Definisco z con Set *)
z[x_, y_] = x + y;
```

```
z[Sqrt[3], 2 Pi]
z[x, y]
```

```
1 + Log[2]
```

```
1 + Log[2]
```

(\* NOTA: se definisco z con SetDelayed, posso usare Set per definire x ed y \*)

```
Clear[z, x, y];
x = 1; y = Log[2];
```

(\* Definisco z con SetDelayed \*)

```
z[x_, y_] := x + y;
z[Sqrt[3], 2 Pi]
z[x, y]
```

```
Sqrt[3] + 2 Pi
```

```
1 + Log[2]
```

⌘ Gli underscore (x\_, y\_, ecc.) nel lhs di una funzione sono importanti:  
essi indicano che x ed y sono parametri formali (non sono valori letterali).

Underscore e' simbolo speciale di input per Blank[] → suggerisce che esso significhi *riempire lo spazio Blank*.

Gli underscore appaiono solo nel lhs di una definizione di funzione (non appaiono mai nel rhs).

⌘ Per la definizione di una funzione, SetDelayed[] e' quasi sempre la scelta giusta.  
Inoltre, ogni parametro formale dovrebbe essere seguito da un Blank.

#### ■ Esercizio 1 pg. 39. L'importanza dell'uso di SetDelayed (non nec)

#### ■ Esercizio 2 pg. 39. Il significato di Blank

Definiamo una funzione in modo incorretto:

```

Clear[f, a];
f[a] = a^2
(* qui abbiamo definito solo f[a] i.e. a e' un valore letterale *)
a^2

```

Ora, valutiamo le espressioni  $f[a]$  ed  $f[b]$ .

```

(* f[a] e' noto al Kernel, f[b] non e' noto al Kernel *)
{f[a], f[b]}                                f[b] non è noto perché nn è una funzione di argomento di variabili
{a^2, f[b]}

```

### DISGRESSIONE:D[] e Derivative[]

```

(* f' e' il simbolo speciale di input per Derivative[1][f] *)
Clear[f, a, x]
{f' == Derivative[1][f],
 f'[a] == Derivative[1][f][a]}          Derivativa ha una sintassi più da operatore (funzione applicata ad un'altra funzione) piuttosto che da funzione.
                                         D → ci chiede la variabile perché è una funzione più generale (si aspetta, per esempio, anche derivate parziali). Insieme a N, sono le due built in più usate.

(* Derivative e' un operatore,
che permette di differenziare simbolicamente la head f,
anche quando f non e' definita . Permette di valutare, ad esempio: *)
{f'[x^2],                                     /. → (replace all) fa un rimpiazzo, cerco f nella parte sinistra e se lo trova lo sostituisce con Sin.
 Sin'[x^2],
 f'[x^2] /. f → Sin}
(* f'[x^2]== Derivative[1][f][x^2] *)

(* Non posso valutare D[f[x],x^2] oppure D[Sin[x],x^2] *)
(* Per ottenere Derivative[1][f][x^2] , devo combinare D e /. . *)
{Derivative[1][f][x^2] === (D[f[x], x] /. x → x^2),
 Derivative[1][Sin][x^2] ===(D[Sin[x], x] /. x → x^2),
 Derivative[1][f /. f → Sin][x^2] === (D[f[x], x] /. {f → Sin, x → x^2}),
 (Derivative[1][f][x] /. {f → Sin, x → x^2}) === (D[f[x], x] /. {f → Sin, x → x^2})}

{True, True}                                    c'è un uguaglianza === tra la prima scrittura e la seconda che però devo avvolgerla tra le parentesi tonde altrimenti non avremo un uguaglianza → replace all verrebbe attivato prima dell'uguaglianza, avremmo una sostituzione (di x^2) che verrebbe calcolata come derivazione.

{f'[x^2], Cos[x^2], Cos[x^2]}
{True, True, True, True}

```

Torniamo al nostro esempio iniziale.

```
Clear[f, a, b];
f[a] = a^2;
{f[a], f'[a], D[f[a], a]}
{f[b], f'[b], D[f[b], b]}
```

$$\{a^2, f[a], 2 a\}$$

$$\{f[b], f'[b], f''[b]\}$$

(\* Verifichiamo meglio cio' che accade, usando Trace \*)

```
{f[a] // FullForm,
f'[a] // FullForm}
D[f[a], a] // Trace
```

```
{f[b] // FullForm,
f'[b] // FullForm}
D[f[b], b] // Trace
```

$$\{\text{Power}[a, 2], \text{Derivative}[1][f][a]\}$$

$$\{\{f[a], a^2\}, \partial_a a^2, 2 a\}$$

$$\{f[b], \text{Derivative}[1][f][b]\}$$

$$\{\partial_b f[b], f'[b]\}$$

Il significato di  $a_$ , nella definizione del lhs della funzione  $f$ , dovrebbe ora essere piu' chiaro.

Ridefiniamo  $f$  in modo corretto, come funzione del parametro formale  $a_$  (non come funzione del valore letterale  $a$ ):

```
Clear[f];
f[a_] := a^2;
{f[a],
f[b]}
```

$$\{a^2, b^2\}$$

### ■ Esercizio 3 pg. 39

Scrivere una funzione che calcoli l'area di un cerchio, dato il suo raggio.

```
Clear[f, x, y, z];
f[r_] := Pi r^2;
{f[1/2], f[1], f[2]}
```

(\* NB: f[ a, b ] non e' definita \*)  
 $f\left[\frac{1}{2}, 1\right]$

 $f\left[\frac{1}{2}, 1\right]$ 

(\* NB: 1 lista  $\leftrightarrow$  1 argomento , quindi f[ lista ] e' definita.  
L'esito dipende dalle proprietà di Times \*)

```
f[{1/2, 1, 2}]
f[r]/. r → {1/2, 1, 2}
Attributes[Times]
```

 $\left\{\frac{\pi}{4}, \pi, 4\pi\right\}$ 
 $\left\{\frac{\pi}{4}, \pi, 4\pi\right\}$ 

```
{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

## 2.3.7 Espressioni composite

Espressioni multiple possono essere piazzate ovunque, separandole con punto-e-virgola;

```
Clear[a]; a = 1 ; b = 2; a + b
```

```
3
```

⌘ L'output di ciascuna espressione seguita da punto-e-virgola viene soppresso.

Questo e' utile ogni volta che l'output di una certa espressione sia, ad esempio, ovvio oppure troppo ingombrante.

⌘ L'input a=1; b=2; a+b costituisce in effetti una sola espressione:

ciascuno dei singoli statement e' parte di una unica espressione avente CompoundExpression[] come head.  
Proviamo a vederlo con FullForm:

```
FullForm[a = 1; b = 2; a + b]
```

```
3
```

⌘ L'output "3" e' dovuto al fatto che CompoundExpression[] viene valutata prima di FullForm[].

Allora, inibiamo la valutazione della CompoundExpression[], avvolgendola in Hold[]:

```
FullForm[Hold[a = 1; b = 2; a + b]]
```

```
Hold[CompoundExpression[Set[a, 1], Set[b, 2], Plus[a, b]]]
```

⌘ Dato che CompoundExpression[] e' una espressione singola, una sequenza di espressioni separate da punto-e-virgola puo' essere piazzata ovunque possa essere piazzata una singola espressione.

L' utilita' di cio' e' permettere uno stile di programmazione che ricorda linguaggi **procedurali** (Fortran).

Per esempio, una funzione, consistente in piu' di una sola linea di codice, puo' essere scritta come segue:

```
f[x_] := (
  firstLine;
  secondLine;
  ....
  lastLine
)
```

⌘ Notiamo le parentesi tonde ( ) attorno al corpo del programma.

Esse sono necessarie perche' il punto-e-virgola ha la precedenza piu' bassa di ogni forma speciale di input in **Mathematica**.

Senza le parentesi, solo la prima riga firstLine andrebbe a fare parte della definizione della funzione f[x].

⌘ Notiamo, inoltre, l'assenza del punto-e-virgola dopo l'ultimo statement lastLine.

Una funzione scritta con molte righe di codice (multi-line function) restituisce il valore dell'ultima espressione

valutata dalla funzione stessa.

La funzione seguente, pertanto, restituisce Null:

```
g[x_] := (
  firstLine;
  secondLine;
  ....
  lastLine;
)
```

⌘ NOTA. Per una definizione di funzione, usare Module oppure Block, evitare l'uso delle parentesi tonde.

### FindRoot[] ed il suo argomento opzionale StepMonitor

⌘ Esistono applicazioni meno ovvie delle espressioni composite.

Per esempio, consideriamo la funzione FindRoot[], solutore iterativo: restituisce una radice (approssimata) a partire da una iterata iniziale (convergenza locale). Di default, FindRoot non restituisce la sequenza di iterate.

#### ? FindRoot

Symbol



FindRoot[f, {x, x<sub>0</sub>}] searches for a numerical root of f, starting from the point x = x<sub>0</sub>.

FindRoot[lhs == rhs, {x, x<sub>0</sub>}] searches for a numerical solution to the equation lhs == rhs.

FindRoot[{f<sub>1</sub>, f<sub>2</sub>, ...}, {{x, x<sub>0</sub>}, {y, y<sub>0</sub>}, ...}] searches for a simultaneous numerical root of all the f<sub>i</sub>.

FindRoot[{eqn<sub>1</sub>, eqn<sub>2</sub>, ...}, {{x, x<sub>0</sub>}, {y, y<sub>0</sub>}, ...}] searches

for a numerical solution to the simultaneous equations eqn<sub>i</sub>.



```
Clear[x];
```

```
Plot[Sin[x] - Cos[x], {x, -3, 5}], ImageSize -> Small]
```

(\* Di default, FindRoot lavora in Precisione-Macchina

i.e. l'opzione WorkingPrecision e' settata a MachinePrecision \*)

```
FindRoot[Sin[x] - Cos[x], {x, 1/2}]
```

Plot -> guarda la funzione e il suo dominio, campiona le x, discretizza il dominio, poi valuta e si formano tanti puntini -> in corrispondenza di ogni puntino viene acceso un pixel, così l'occhio vede una linea continua (calcola in maniera automatica quanti punti servono per mostrare una linea continua).

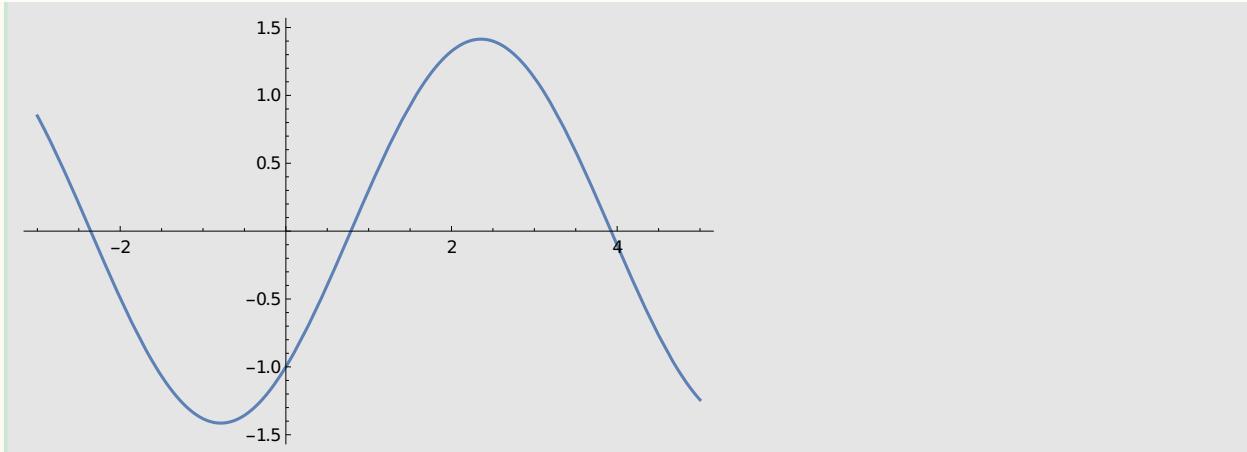
```
FindRoot[Sin[x] - Cos[x], {x, 0.5}]
```

Find -> built in che dipende dal comportamento dell'input.

Solve -> vede come argomento un'equazione i cui coefficienti sono tutti esatti, allora innesca un metodo di soluzione algebrico, altrimenti se c'è un numero reale innesca un metodo numerico di soluzione

Find Root -> numerico, restituisce SOLO una funzione numerica in double precision. Cerca la root di un'equazione. Dobbiamo dargli un info di dove più o meno sta questa soluzione. Ci dà un'approssimazione. Problema difficile -> un metodo iterativo è l'unico modo per trovare la soluzione.

esempio: quando abbiamo una funzione non lineare, meglio PLOTTARLA!



```
{x → 0.785398}
```

```
{x → 0.785398}
```

⌘ L'output ottenuto non fornisce indicazioni sulla sequenza di iterate che e' stata creata per arrivare a tale risposta.

Possiamo usare Print[ ] e creare l'espressione composta :

```
(Print[x]; Sin[x] - Cos[x]);
```

⌘ Ogni volta che FindRoot valuta tale espressione, il valore corrente di x viene stampato da Print[ ].

```
Clear[f, x];
f[x_?NumberQ] := (Print[x]; Sin[x] - Cos[x]);
FindRoot[f[x], {x, 0.5}]
(* La stampa di ogni x viene , pero', ripetuta 3 volte *)
```

0.5

0.5 NumberQ → vuol dire che la f viene valutata solo x se è un argomento con cui NumberQ mi restituisce TRUE

0.5 Numer = numeri interi, reali, machine precision, ecc...

0.793408

0.793408

0.793408

0.785398

0.785398

0.785398

0.785398

0.785398

{ $x \rightarrow 0.785398$ }

⌘ Meglio usare l'argomento opzionale StepMonitor di FindRoot

E' come se l'espressione (composita) di cui si cerca la radice sia, appunto ( $\text{Print}[x]; \text{Sin}[x]-\text{Cos}[x]$ );

```
FindRoot[Sin[x]-Cos[x], {x, 0.5}, StepMonitor :> Print["Step to x = ", x]]
```

Step to  $x = 0.793408$

Step to  $x = 0.785398$  Se voglio n stampe, attivo una option che monitora gli step (step monitor). di default è disabilitata.

Step to  $x = 0.785398$  Se scrivo :> diventa automatica quel simbolo :—>

{ $x \rightarrow 0.785398$ }

⌘ StepMonitor e' una delle Options di FindRoot

? StepMonitor

Symbol



StepMonitor is an option for iterative numerical computation functions that gives an expression to evaluate whenever a step is taken by the numerical method used.



Options[FindRoot]

```
{AccuracyGoal → Automatic, Compiled → Automatic, DampingFactor → 1,
 Evaluated → True, EvaluationMonitor → None, Jacobian → Automatic,
 MaxIterations → 100, Method → Automatic, PrecisionGoal → Automatic,
 StepMonitor → None, WorkingPrecision → MachinePrecision}
```

di default, l'opzione di stampa è settata "none" (non deve stampare)

⌘ NOTA. Per sovrascrivere il default (None) dell'argomento opzionale StepMonitor, devo usare la regola differita :> (RuleDelayed), altrimenti "x" nella Print viene valutato immediatamente (prima che il metodo abbia iniziato ad assegnare valori numerici ad x) e quindi viene restituita una sola Print del simbolo "x" stesso .

```
FindRoot[Sin[x]-Cos[x], {x, 0.5}, StepMonitor :> Print["Step to x = ", x]]
```

Step to  $x = x$

{ $x \rightarrow 0.785398$ }

⌘ Un buon riferimento e' Monitoring and Selecting Algorithms, penultima sezione nel tutorial Numerical Operations on Functions

```
Hyperlink["Tutorial sul Monitoraggio numerico",
 "https://reference.wolfram.com/language/tutorial/NumericalOperationsOnFunctions.
 html#25086"]
```

Tutorial sul Monitoraggio numerico

#### ■ Esercizio 1 pg. 41

Quale e' la rappresentazione interna della espressione `z=1;` ?

```
ClearAll[z];
z = 1;
FullForm[z = 1;]
```

Null

Dobbiamo usare Hold per capirlo:

<code>FullForm[Hold[z = 1;]]</code>	metto una Hold per vedere che z è settata a 1
<code>Hold[CompoundExpression[Set[z, 1], Null]]</code>	

#### ■ Esercizio 2 pg. 41

### 2.3.8 Liste

### 2.3.9 Regole

### 2.3.10 Controllo del flusso (non nec)

### 2.3.11 Trappole sintattiche per l'inconsapevole

### 2.3.12 Informazioni sui simboli (non nec)

## 2.3.7 Espressioni composite

### 2.3.8 Liste

Le liste costituiscono la struttura-dati di base in *Mathematica*.

Una lista è usata per raggruppare espressioni in un ordine particolare.

⌘ Una lista è delimitata da parentesi graffe { }.

```
{1, x, i + j}
(* questa e' una lista di tre espressioni *)
FullForm[%]
(* Possiamo vederne la forma interna con FullForm *)

{1, x, i + j}
```

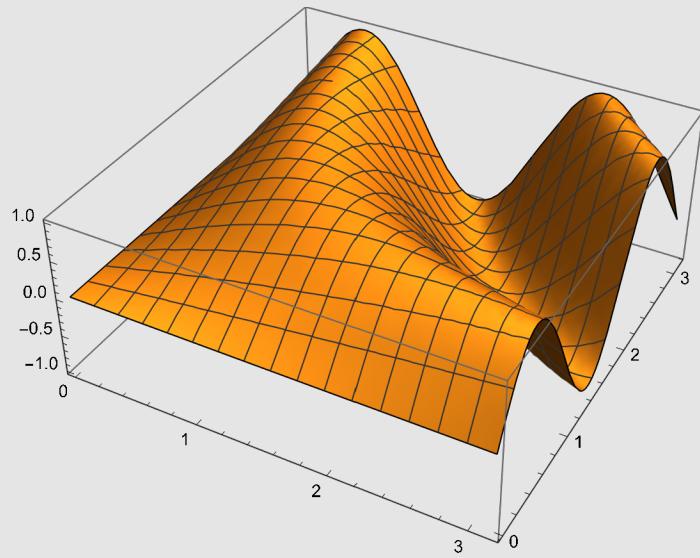
```
List[1, x, Plus[i, j]]
```

⌘ La head List non comporta l'effettuazione di alcuna valutazione  
(essa ha semplicemente interessanti forme speciali di input e di output).

⌘ Molte delle funzioni built-in richiedono che certi parametri siano raggruppati in liste.

Ad esempio, nell'espressione Plot3D[ ] qui sotto le liste sono usate per raggruppare ogni variabile indipendente con l'intervallo di plot desiderato:

```
Plot3D[{Cos[x y], Sin[x y]}, {x, 0, Pi}, {y, 0, Pi}]
```



⌘ Esiste un altro insieme di delimitatori che si incontra abbastanza spesso, quando si lavora con le liste:

doppi parentesi quadre [[ ]],  
che vengono usate per estrarre parti di una lista

```
mylist = {1, x, i + j};
(* estraggo il terzo elemento della lista *)
mylist[[3]]
Part[mylist, 3]
```

i + j

i + j

⌘ Le liste, pertanto, sono analoghe agli array di linguaggi procedurali, quali Fortran e C.  
Notiamo che le liste in *Mathematica* usano una indicizzazione che parte da 1, come in Fortran (non da 0, come in C).

⌘ Dato che le liste stesse sono espressioni, esse possono annidarsi in modo arbitrario.  
Per convenzione, le liste annidate rettangolari sono usate per rappresentare matrici (memorizzare per righe: ogni riga e' una sottolista):

```
(* s e' una lista annidata rettangolare, che rappresenta una matrice 3x2 *)
s = {{a, b}, {c, d}, {e, f}}; MatrixForm[s]
(* gli elementi di s sono a loro volta liste; s[[1]] e' la riga 1 della matrice *)
s[[1]]
(* le sottoliste sono indicizzate a loro volta;
pertanto %[[2]] estrae il secondo elemento della riga 1 di s *)
%[[2]]
(* estraggo l'elemento 1,2 di s, con un solo comando *)
s[[1, 2]]
```

$$\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}$$

{a, b}

b

b

⌘ Le liste sono pervasive in *Mathematica*, per cui esistono moltissime built-in per la manipolazione di liste.

#### ■ Esercizio 1 pg. 42

Quale sotto-indice e' necessario usare per estrarre l'elemento **c** dalla lista che segue?

E per estrarre l'elemento **e** ?

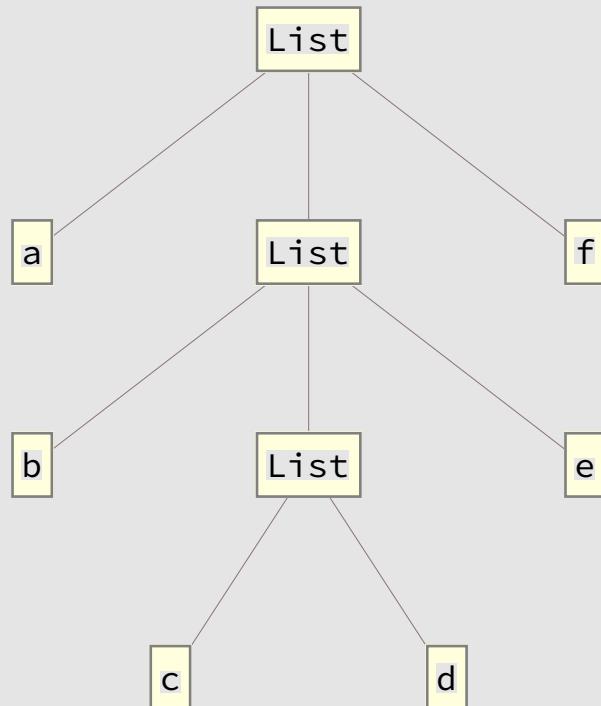
```
Clear[a, b, c, d, e, f]
mialista = {a, {b, {c, d}, e}, f}
```

```
{a, {b, {c, d}, e}, f}
```

```
{c == mialista[[2, 2, 1]], e == mialista[[2, 3]]}
```

```
{True, True}
```

```
mialista // TreeForm
```



### 2.3.9 Regole

### 2.3.10 Controllo del flusso (non nec)

### 2.3.11 Trappole sintattiche per l'inconsapevole

# Colors and Styles

In *Mathematica*, you can handle various things (not just numbers), e.g., colors.

You can refer to common colors by their names.

```
{Red, Green, Blue, Purple, Orange, Black}
```

```
{, , , , , }
```

You can do operations on colors.

- **ColorNegate** gives the complementary color of a specified color, defined by computing  $1 - \text{value}$  for each RGB component.

If you negate Red, Green, Blue, you get Cyan, Magenta, Yellow.

```
(* Red==RGBColor[1,0,0], Cyan==RGBColor[0,1,1] *)
{Red, Cyan, ColorNegate[Cyan] == Red}
(* Green==RGBColor[0,1,0], Magenta==RGBColor[1,0,1] *)
{Green, Magenta, ColorNegate[Magenta] == Green}
(* Blue==RGBColor[0,0,1], Yellow==RGBColor[1,1,0] *)
{Blue, Yellow, ColorNegate[Yellow] == Blue}

{, , True}
{, , True}
{, , True}
```

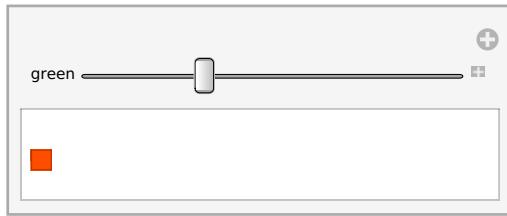
- **Blend** blends a list of colors together.

```
Blend[{Yellow, Pink, Green}]
(* Blend needs 1 compulsory argument *)
(* NO: Blend[Yellow,Pink,Green] *)
(* NO: Blend[] *)
```



You can specify a color by saying how much Red, Green, Blue (**RGBColor**) it contains.

```
(* From Red to Yellow *)
(* Note 1: {Red==RGBColor[1,0.,0], Yellow==RGBColor[1,1.,0]} *)
(* Note 2: Values outside [0,1] are clipped *)
Table[ green, {green, 0, 1, 1/20.} ]
Table[ RGBColor[1, green, 0], {green, 0, 1, 1/20.} ]
Manipulate[ RGBColor[1, green, 0], {green, 0, 1, 1/20.} ]
{0., 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4,
 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.}
{█, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █}
```



You can specify a colors in terms of **Hue**

- **hue** is a pure color.

Colors of different hues are often arranged around a **color wheel**

(Newton 1642–1727 UK, Goethe 1749–1832 D, Chevreul 1786–1889 F, Munsell 1858–1918 USA, Itten 1888–1967 CH,...)

RGB values for a particular Hue are given by a math formula.

- **Hue[ ]** uses a combination of tint/saturation/brightness.

```
{Hue[0.5], Hue[1/2] == Hue[0.5]}
Table[ Hue[ c ], {c, 0, 1, 1/20} ]
{█, True}
{█, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █}
```

**RandomColor** lets you pick a random color

`RandomInteger[10]` generates a random integer up to 10.

For a random color you do not have to specify a range, so you can just write `RandomColor[]`, without any explicit input.

```
SeedRandom[8];
(* RandomColor[] *)
Table[ RandomColor[], 30 ]
{█, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █, █}
```

**Blending** together random colors usually gives something muddy:

```
SeedRandom[8];
b8 = Blend[ Table[RandomColor[], 20] ]
FullForm[b8]
█
RGBColor[0.4081000861042466`, 0.5404537481816792`, 0.5226400839157853`]
```

You can use colors in all sorts of places (e.g. **Style**).

For example, you can give a **Style** to output with colors.

```
Style[1000, Red]
(* Coloro 10 interi, generati random in [0,1000], con colori random *)
SeedRandom[8];
Table[
  Style[ RandomInteger[1000], RandomColor[] ],
  10]
{9, 461, 680, 137, 345, 123, 844, 453, 969, 772}

(* Nota: SeedRandom influenza tutti i generatori Random.
   Per controllarli singolarmente, li valutiamo singolarmente : *)
SeedRandom[8];
tri = Table[ RandomInteger[1000], 10]
```

```
SeedRandom[8];
trc = Table[RandomColor[], 10]

Table[ Style[tri[[k]], trc[[k]]], {k, 1, 10}]
{9, 205, 525, 519, 159, 636, 351, 585, 355, 973}
```

```
{█, █, █, █, █, █, █, █, █, █}
{9, 205, 525, 519, 159, 636, 351, 585, 355, 973}
```

- Another form of styling is **size**.

You can specify a font size in **Style**.

```
(* Show x styled in 30-point type *)
{Style[x, 30],
 Style[x, Bold, 30],
 Style[x, Italic, 30],
 Style[x, Italic, 30, FontFamily -> "Times"]}

{X, X, X, X}

(* Number 100 in different sizes*)
Table[ Style[100, n], {n, 8, 30, 2} ]
{100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100}
```

- You can combine **color** and **size** styling.

```
(* x in 25 random colors and sizes*)
SeedRandom[2];
Table[
 Style[ x , RandomColor[], RandomInteger[30] ],
 5]
{x, x, x, X, X}
```

## Vocabulary

Red, Green, Blue, Yellow, Orange, Pink, Purple, ...	colors
RGBColor[0.4,0.7,0.3]	red, green, blue color
Hue[0.8]	color specified by hue
RandomColor[]	randomly chosen color
ColorNegate[Red]	negate a color (complement)
Blend[{Red,Blue}]	blend a list of colors
Style[x,Red]	style with a color
Style[x,20]	style with a size
Style[x,20,Red]	style with a size
<b>\$FontFamilies</b>	
Import, Export, <b>\$ImportFormat</b> , <b>\$ExportFormat</b>	
GrayLevel	

## Exercises

### 7.1 Make a list of red, yellow and green.

**7.2** Make a red, yellow, green Column (traffic light).

**7.3** Compute the negation of the color orange.

**7.4** Make a list of colors with hues varying from 0 to 1 in steps of 1/50 (0.02).

**7.5** Make a list of colors with maximum Red and Blue, but with Green varying from 0 to 1 in steps of 1/20 (0.05).

**7.6** Blend the colors pink and yellow.

**7.7** Make a list of colors obtained by **blending** Yellow with hues from 0 to 1 in steps of 1/20 (0.05).

**7.8** Make a list of **numbers** from 0 to 1 in steps of 1/10 (0.1), where each number has a hue equal to its value.

**7.9** Make a **Purple** swatch (= small square used to display a color) of size 100.

**7.10** Make a list of **Red** swatches with sizes from 10 to 100 in steps of 10.

**7.11** Display the number 789 in **Red** at size 100.

**7.12** Make a list of the first 9 squares, in which each value is styled at its size (**MapThread**).

**7.12bis** Make a list of the first 4 even numbers (starting with 4) **squared**, in which each value is styled at its size (**MapThread**).

**7.13** Use **Part** and **RandomInteger** to make a length-100 list in which each element is randomly Red, Yellow or Green.

**7.14** Use **Part** to make a list of the first 50 digits in  $2^{1000}$  (eliminate 0|1), in which each digit has size equal to 3 times its value (Cases, Except, Alternatives).

**+7.1** Create a Column of colors with Hue varying from 0 to 1 in steps of 1/20 (0.05).

**+7.2** Make a list of colors varying from Red to Green (Blue=0), with green components **g** varying from 0 to 1 in steps of 1/20 (0.05), and with red components **1-g**.

**+7.3** Create a list of colors with no Red nor Blue, and with Green varying from 0 to 1, and back down to 0, in increments of 1/10 (the 1 should not be repeated).

**+7.4** Blend the color Red and its negation.

**+7.5** Blend a list of colors with Hue from 0 to 1 in increments of 1/10 (0.1).

**+7.6** Blend the color Red with White, then blend it again with White (**NestList**) .

**+7.7** Make a list of 50 random colors.

**+7.8** Make a 2-entries Column (2 x N, but it will not be a matrix, due to the Column wrapping) for each number 1 through 5, with the number rendered first in Red then in Green.

**+7.9** Make columns of the numbers 1 through 10, rendered as plain/ bold / italic in each column (**TableForm**, **Transpose**,**Thread**).

## Q & A

Are there other ways to specify colors than **RGBColor** or **Hue**?

Yes, e.g. other color models, like **CMYKColor**[cyan, magenta, yellow, black] (it refers to the cyan, magenta, yellow, black inks used in printers), or **GrayLevel** that represents shades of gray, with GrayLevel[0] being Black and GrayLevel[1] being White.

```
{GrayLevel[0] == Black, GrayLevel[1] == White}
{True, True}
```

Device-independent CIE color models are also available, like **LABColor** and **XYZColor** (CIE : Commission Internationale de l'Eclairage, International Commission on Illumination).

## Tech Notes

Examples of other ways to specify colors (than RGB or Hue).

**LABColor**[L,  $\alpha$ ,  $\beta$ ] or **LABColor**[{L,  $\alpha$ ,  $\beta$ }],

where L is lightness (brightness)

and  $\alpha$ ,  $\beta$  are color components (respectively, Green to Magenta, Blue to Yellow) .

You can specify the Optional argument  $\omega$  opacity: **LABColor**[L,  $\alpha$ ,  $\beta$ ,  $\omega$ ]; default is  $\omega=1$ .

```
{LABColor[1, -1, 1],
 LABColor[1, 1, -1],
 LABColor[1, 0, -1],
 LABColor[1, 0, 1]}
```

{, , , }

**XYZColor**[x, y, z], where **x** is color (combination of Red/Green), **y** is Luminance (visually perceived brightness), **z** is color (Blue).

```
{XYZColor[0, 1, 0],
 XYZColor[1, 1/2, 0],
 XYZColor[1, 1, 0],
 XYZColor[0, 0, 1]}
```

{, , , }

You can specify named HTML colors (e.g. `RGBColor["aqua"]`) as well as hex colors (e.g. `RGBColor["#00ff00"]`)

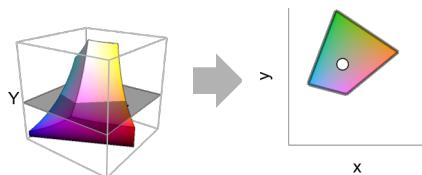
```
{RGBColor["aqua"], RGBColor["aqua"] == Cyan,
 RGBColor["#00ff00"], RGBColor["#00ff00"] == Green}
{, True, , True}
```

**ChromaticityPlot** and **ChromaticityPlot3D** plot lists of colors in color space.

**ChromaticityPlot** is used to visualize one or several colors in an image.

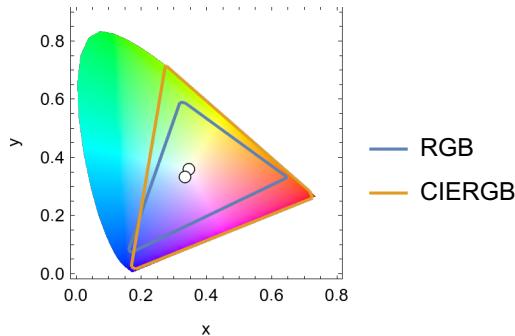
**ChromaticityPlot**[*colspace*] plots a 2D slice of the color space *colspace*

(i.e., convert the color coordinates in *colspace* to coordinates in *refcolspace* color space, and displays a slice given by constant luminance 0.01).



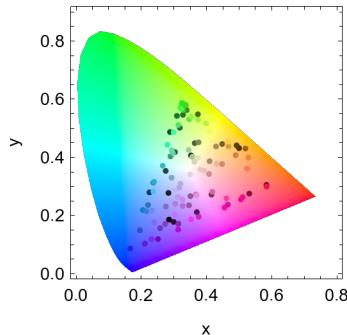
→ It can be used to compare to color space models.

```
(* Confronto la gamma dei due spazi colore RGB e CIERGB
(CIE:Commission Internationale de l'Eclairage,
International Commission on Illumination) *)
ChromaticityPlot[{"RGB", "CIERGB"}, ImageSize → Small]
```



```
(* Visualizzo una lista di colori RGB random *)
```

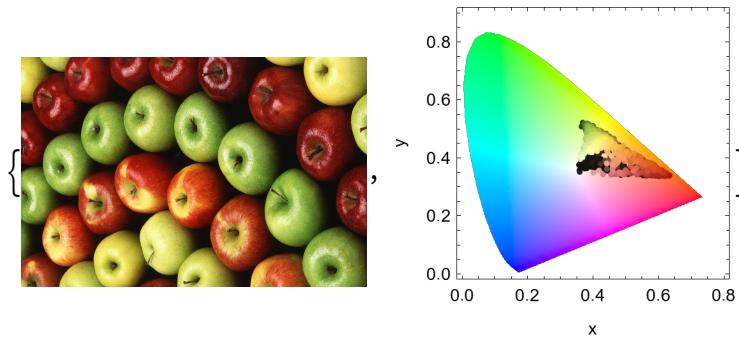
```
SeedRandom[8];
ChromaticityPlot[RandomColor[100], ImageSize → Small]
```



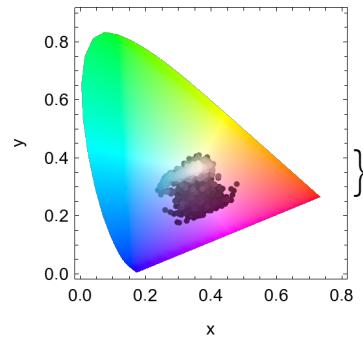
ChromaticityPlot[image] plots the pixels of *image* as individual colors.

→ It can be used to visualize the pixels of an image.

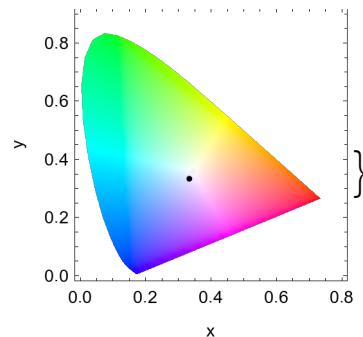
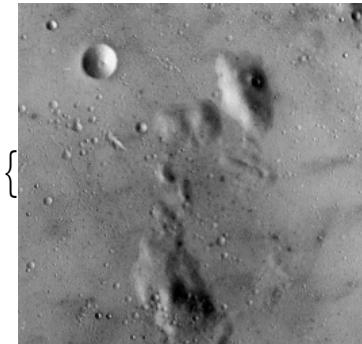
```
(* img=ExampleData[{"TestImage", "Peppers"}]; *)
img = ExampleData[{"TestImage", "Apples"}];
{Image[img, ImageSize → Small],
ChromaticityPlot[img, ImageSize → Small]}
```



```
img = ExampleData[{"TestImage", "Tree"}];  
{Image[img, ImageSize → Small],  
 ChromaticityPlot[img, ImageSize → Small]}
```



```
(* img=ExampleData[{"TestImage","Bridge"}]; *)  
img = ExampleData[{"TestImage", "Moon"}];  
{Image[img, ImageSize → Small],  
 ChromaticityPlot[img, ImageSize → Small]}
```



You can set lots of other style attributes in *Mathematica*, like **Bold**, **Italic** and **FontFamily**.

## 2.3.9 Regole

La forma speciale di input  $a \rightarrow b$  e' detta Regola (Rule).

⌘ Una Regola, da sola, non e' altro che un container per una coppia di espressioni (con una forma speciale di input/output), ma ci sono molte funzioni che si aspettano Regole come argomenti.

⌘ La piu' comune di tali funzioni e' ReplaceAll[]

? ReplaceAll

*expr /. rules* applies a rule or list of rules in an attempt to transform each subpart of an expression *expr*. >>

ReplaceAll[espressione,  $a \rightarrow b$ ] rimpiazza con "b" ogni occorrenza di "a" in "espressione".

ReplaceAll[x + y^2, x → w]

w + y<sup>2</sup>

ReplaceAll[] e' cosi' comune che esiste una forma speciale (/. ) di input per tale funzione

expression /. rule

(\* equivalente a ReplaceAll[ expression, rule ] \*)

expression /. rule

x + y^2 /. x → w

w + y<sup>2</sup>

ReplaceAll[] puo' rimpiazzare una espressione arbitraria con un'altra espressione arbitraria.

Si noti, pero', che la sostituzione della Rule e' puramente sintattica, non e' algebrica.

x^2 + x^4 /. x^2 → 1+w

(\* ReplaceAll non cerca di sostituire x^4,  
sostituisce solo x^2 , ossia Power[x,2] \*)

1 + w + x<sup>4</sup>

```
(* La regola di sostituzione e' definita solo per Power[x,2] *)
x^2 + x^4 // FullForm
x^2 → 1+w // FullForm

Plus[Power[x, 2], Power[x, 4]]
```

```
Rule[Power[x, 2], Plus[1, w]]
```

(\* Per ottenere una sostituzione algebrica,  
posso rendere pattern l'esponente (come vedremo meglio piu' avanti).  
Ad esempio: \*)

```
x^2 + x^4 /. x^n_ → (1+w)^(n/2)

1+w+(1+w)^2
```

⌘ Queste che seguono sono altre proprieta' di ReplaceAll[]

```
x^2 + x^4 /. { x^2 → 1+w, x^4 → (1+w)^2 }
(* Il secondo argomento di una ReplaceAll puo' essere una lista di Regole *)
1+w+(1+w)^2
```

```
(* L'operatore ReplaceAll e' associativo a sinistra,
ossia si va da Sx verso Dx *)
x + x^2 /. x → w^2 /. w^2 → z
(x + x^2 /. x → w^2 /. w^2 → z) === ((x + x^2 /. x → w^2) /. w^2 → z)

(* Passo 1: x + x^2 /. x → w^2
Otteniamo w^2+w^4 *)

(* Passo 2: w^2+w^4 /. w^2 → z
Otteniamo w^4+z *)
{x + x^2 /. x → w^2 ,
w^2+w^4 /. w^2 → z}

w^4 + z
```

True

{w<sup>2</sup> + w<sup>4</sup>, w<sup>4</sup> + z}

```
(* Se l'operatore ReplaceAll fosse stato associativo a destra : *)
(* Passo 1: x + x^2 /. w^2 → z
Otteniamo x+x2 *)

(* Passo 2: x+x2 /. x → w^2
Otteniamo w2+w4 *)
{x + x^2 /. w^2 → z,
x+x2 /. x → w^2}

{x+x2, w2 + w4}
```

⌘ Un altro impiego comune per le Regole e' nello specificare Opzioni ad una funzione; tali argomenti opzionali sono detti "argomenti-con-nome" (named arguments) ed hanno la forma nome → valore (name → value).

⌘ Questo e' in contrasto con gli "argomenti posizionali" ordinari, i cui nomi sono inferiti dalla loro posizione (nella sequenza di argomenti alla funzione stessa).

⌘ Dato che le Opzioni portano con se' il loro nome, esse non devono apparire in alcun ordine predeterminato (anzi, esse possono anche non apparire affatto).

⌘ Le Opzioni sono, pertanto, utili per funzioni di molti parametri (quali, ad esempio, le funzioni di grafica), oppure per parametri che vengono usati poco frequentemente.

⌘ Le Opzioni vengono sempre specificate dopo tutti gli argomenti posizionali.

⌘ La built-in Options[ ] restituisce la lista di tutti gli argomenti opzionali (e dei loro valori di default) di una data funzione:

```
? FactorInteger
(* La funzione FactorInteger ha un argomento opzionale GaussianInteger settato,
di default, a Falso *)
Options[FactorInteger]
(* Il numero primo 397 non puo' essere
fattorizzato sull'insieme Z degli interi,
ma puo' essere fattorizzato rispetti agli interi Gaussiani *)
PrimeQ[397]
FactorInteger[397]
FactorInteger[397, GaussianIntegers → True]
(* Nota. (Z,+,x) e' un anello commutativo. Non e' un campo perche' solo ±
1 hanno elemento inverso. Ad esempio,
in Z non esiste inverso di 2, che sarebbe 2^-1 = 1/2 *)
```

Symbol

*i*

FactorInteger[n] gives a list of the prime factors of the integer  $n$ , together with their exponents.

FactorInteger[n, k] does partial factorization, pulling out at most  $k$  distinct factors.

▼

{GaussianIntegers → False}

True

{{{397, 1}}}

$\{-i, 1\}, \{6 + 19i, 1\}, \{19 + 6i, 1\}$

⌘ Se si devono effettuare molte chiamate ad una funzione, usando sempre le stesse specifiche di argomenti opzionali, conviene cambiare i valori di default per tali argomenti opzionali.

Questo si puo' fare con SetOptions[ ]

```
(* Altero il valore di default per l'Opzione
 GaussianIntegers della funzione FactorInteger *)
SetOptions[FactorInteger, GaussianIntegers → True]
(* Ora FactorInteger[] lavora sempre con l'opzione GaussianIntegers→True *)
FactorInteger[397]

{GaussianIntegers → True}
```

```
{\{-i, 1\}, \{6 + 19 i, 1\}, \{19 + 6 i, 1\}}
```

```
(* ?Plot *)
Options[Plot]
?AspectRatio
(* Tra le Options di Plot c'e' AspectRatio *)

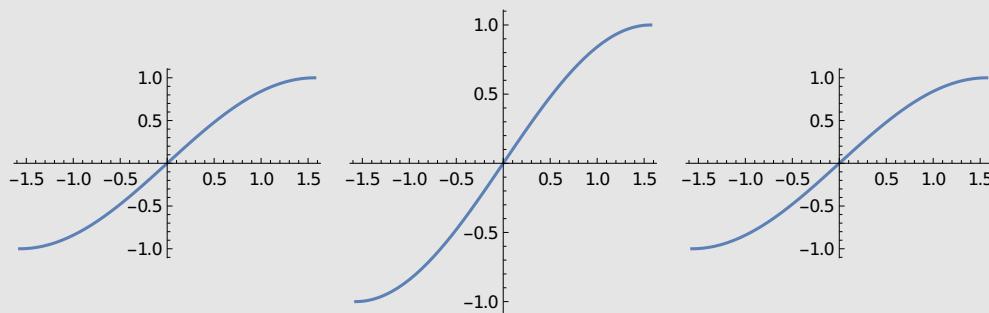
{AlignmentPoint -> Center, AspectRatio ->  $\frac{1}{GoldenRatio}$ , Axes -> True,
AxesLabel -> None, AxesOrigin -> Automatic, AxesStyle -> {}, Background -> None,
BaselinePosition -> Automatic, BaseStyle -> {}, ClippingStyle -> None,
ColorFunction -> Automatic, ColorFunctionScaling -> True, ColorOutput -> Automatic,
ContentSelectable -> Automatic, CoordinatesToolOptions -> Automatic,
DisplayFunction -> $DisplayFunction, Epilog -> {}, Evaluated -> Automatic,
EvaluationMonitor -> None, Exclusions -> Automatic, ExclusionsStyle -> None,
Filling -> None, FillingStyle -> Automatic, FormatType -> TraditionalForm,
Frame -> False, FrameLabel -> None, FrameStyle -> {}, FrameTicks -> Automatic,
FrameTicksStyle -> {}, GridLines -> None, GridLinesStyle -> {}, ImageMargins -> 0.,
ImagePadding -> All, ImageSize -> Automatic, ImageSizeRaw -> Automatic,
LabelingSize -> Automatic, LabelStyle -> {}, MaxRecursion -> Automatic,
Mesh -> None, MeshFunctions -> {#1 &}, MeshShading -> None, MeshStyle -> Automatic,
Method -> Automatic, PerformanceGoal -> $PerformanceGoal,
PlotLabel -> None, PlotLabels -> None, PlotLayout -> Automatic,
PlotLegends -> None, PlotPoints -> Automatic, PlotRange -> {Full, Automatic},
PlotRangeClipping -> True, PlotRangePadding -> Automatic,
PlotRegion -> Automatic, PlotStyle -> Automatic, PlotTheme -> $PlotTheme,
PreserveImageOptions -> Automatic, Prolog -> {}, RegionFunction -> (True &),
RotateLabel -> True, ScalingFunctions -> None, TargetUnits -> Automatic,
Ticks -> Automatic, TicksStyle -> {}, WorkingPrecision -> MachinePrecision}
```

Symbol i

AspectRatio is an option for Graphics and related  
functions that specifies the ratio of height to width for a plot.



```
SetOptions[Plot, ImageSize → Small];
(* Setto la Option ImageSize per ogni Plot successiva *)
plot1 = Plot[Sin[x], {x, -Pi/2, Pi/2}];
(* La prima Plot qui sopra usa il valore di default AspectRatio→ $\frac{1}{GoldenRatio}$  *)
SetOptions[Plot, AspectRatio → 1];
plot2 = Plot[Sin[x], {x, -Pi/2, Pi/2}];
(* La seconda Plot qui sopra usa il valore assegnato AspectRatio → 1 *)
SetOptions[Plot, AspectRatio → 1/GoldenRatio];
plot3 = Plot[Sin[x], {x, -Pi/2, Pi/2}];
(* La terza Plot qui sopra usa il
valore riassegnato al default AspectRatio→ $\frac{1}{GoldenRatio}$  *)
GraphicsRow[{plot1, plot2, plot3}]
```



## Programmazione basata su Regole

Nella programmazione basata su regole (Rule-Based), il programmatore scrive un insieme di regole, che specificano quali trasformazioni devono essere applicate ad una certa espressione (incontrata durante la soluzione di un problema).

Il programmatore non deve specificare l'ordine in cui tali regole devono essere eseguite: il sistema di programmazione (che sta sotto tale tipo di programmazione) lo determina da solo.

La programmazione basata su regole e' un modo naturale di implementare calcoli matematici, dato che la matematica (simbolica) essenzialmente consiste nell'applicare regole di trasformazione ad espressioni (e.g. regole di differenziazione, tabelle di integrali).

Le capacita' versatili del procedimento di match-di-pattern, in *Mathematica*, fa sì che la programmazione basata su regole sia il paradigma di programmazione di elezione (per i programmatori di *Mathematica*).

### ■ 6.1. Pattern \*

#### □ 6.1.1 Che cosa e' un Pattern

Un pattern e' una espressione in *Mathematica* che rappresenta una intera classe di espressioni.

Il pattern piu' semplice e' il Blank `_` singolo, che rappresenta qualsiasi espressione.

Un altro esempio e' `_f`, che rappresenta qualsiasi espressione avente `f` come Head.

Abbiamo gia' usato pattern quali i due qui sopra, come parametri formali nella definizione di funzioni; esamineremo il loro uso in maggiore dettaglio nel paragrafo 6.2.

I pattern possono essere usati da una varietà di funzioni built-in, per alterare la struttura di espressioni.

Ad esempio, una regola di sostituzione (Replacement Rule) puo' avere un pattern nella sua componente sinistra (left-hand side): `/lhs → rhs`

Definiamo una espressione "expr":

```
expr = 3 a + 4.5 b^2; expr // FullForm
```

```
Plus[Times[3, a], Times[4.5` , Power[b, 2]]]
```

La regola (Rule) che segue eleva al quadrato ogni numero reale nella espressione expr

```
(* expr = 3 a + 4.5 b^2 *)
expr /. x_Real → x^2
(* l'unico reale in expr   e'   4.5   *)
{3 a + 4.5 b^2, 3 a + 20.25 b^2}
```

La regola (Rule) che segue eleva al quadrato ogni numero intero nella espressione expr (pertanto, anche l'esponente di `b`)

```
(* expr = 3 a + 4.5 b^2 *)
expr /. x_Integer → x^2
```

$$\{3 a + 4.5 b^2, 9 a + 4.5 b^4\}$$

Si deve prestare attenzione a quanto si chiede ... perche' si potrebbe ottenerlo (e magari, pur essendo un output corretto, non e' quanto ci si aspettava)!

```
(* expr = 3 a + 4.5 b^2 *)
expr /. x_Symbol → x^2

Plus2[Times2[3, a2], Times2[4.5, Power2[b2, 2]]]
```

Nell'esempio qui sopra, l'esito della sostituzione genera una espressione priva di utilita' e/o significato.  
Idem nell'esempio qui sotto:

```
(* expr = 3 a + 4.5 b^2 *)
symbexpr = expr /. x_Symbol → x^2;
symbexpr /. {a → 3, b → 2}

Plus2[Times2[3, 9], Times2[4.5, Power2[4, 2]]]
```

$\text{Power}^2[4, 2]$  non e' valutabile ed e' diverso da  $\text{Power}[4, 2]^2$  (e lo stesso vale per Times):

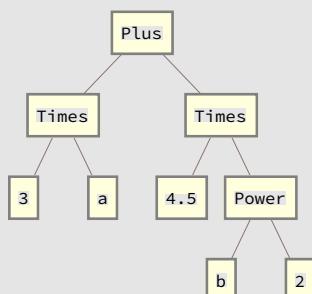
```
(* Power2[4, 2]//FullForm *)
(* Power[4, 2]2//FullForm *)

Power2[4, 2] === Power[4, 2]2
(* Se si vuole eseguire la comparazione con Equal, invece che SameQ,
e si vuole ottenere sempre un Booleano, si puo' usare TrueQ *)
(* TrueQ[Power2[4, 2] == Power[4, 2]2] *)
```

False

Vediamo degli esempi per mostrare che un pattern puo' combaciare (to match) con qualsiasi parte di una espressione, anche con una Head.

```
expr = 3 a + 4.5 b^2;
TreeForm[expr, ImageSize → Small]
```



```
(* match con la Head Plus *)
(* expr = 3 a + 4.5 b^2 *)
expr /. Plus → Times
```

$13.5 a b^2$

```
(* nessuna sostituzione perche' x non e' presente in expr *)
(* expr = 3 a + 4.5 b^2 *)
expr /. x → x^2
```

$3 a + 4.5 b^2$

```
(* match con a : sostituisco a con x^2 *)
(* expr = 3 a + 4.5 b^2 *)
expr /. a → x^2
```

$4.5 b^2 + 3 x^2$

```
(* match con b : sostituisco b con x^2 , per cui b^2 diventa x^4 *)
(* expr = 3 a + 4.5 b^2 *)
expr /. b → x^2
```

$3 a + 4.5 x^4$

```
(* no match : {a,b} non e' in expr *)
(* expr = 3 a + 4.5 b^2 *)
expr /. {a, b} → x^2
(* no match : {a,b} non e' in expr *)
expr /. {a, b} → {x^2, x^2}
```

$3 a + 4.5 b^2$

$3 a + 4.5 b^2$

```
(* match con a , b *)
(* expr = 3 a + 4.5 b^2 *)
expr /. {a → x^2, b → x^2}
```

$3 x^2 + 4.5 x^4$

Gli esempi precedenti mostrano che un pattern puo' combaciare (to match) con qualsiasi parte di una espressione, anche con una Head.

Gli stessi pattern sono espressioni; in pratica, a qualsiasi parte di un pattern puo' essere dato un nome temporaneo, per permettere ad una regola (Rule) di estrarre e manipolare parti di una espressione. Questi nomi temporanei sono detti **variabili — pattern (pattern variables)**.

Costruiamo un esempio in cui useremo una variabile-pattern; definiamo l' espressione test :

```
Clear[expr, f, g, a, b];
test =  $\frac{f[a] + g[b]}{f[a, b]}$ ;
```

Nell'esempio che segue, la variabile - pattern **expr\_f** fa riferimento a **qualsiasi espressione expr con Head f** .

Pertanto **expr\_f** combacia con  $f[a]$  ed anche con  $f[a, b]$ .

La regola di sostituzione qui e'  $expr_f \rightarrow expr^2$  .

Ne segue che ad  $f[a]$  viene sostituito  $f[a]^2$  .

Analogamente  $f[a,b]$  viene rimpiazzato con  $f[a, b]^2$

```
test =  $\frac{f[a] + g[b]}{f[a, b]}$ ;
test /. expr_f → expr^2
(* Possiamo scrivere anche cosi' *)
test /. t_f → t^2
```

$$\frac{f[a]^2 + g[b]}{f[a, b]^2}$$

$$\frac{f[a]^2 + g[b]}{f[a, b]^2}$$

Nell'esempio che segue, la variabile-pattern **f[x\_]** fa riferimento a qualsiasi espressione avente Head **f** ed in cui **f** sia funzione di **una sola** variabile **x** .

Pertanto **f[x\_]** combacia con  $f[a]$ .

La regola di sostituzione qui e'  $f[x_] \rightarrow x^2$  .

Ne segue che  $f[a]$  viene sostituito con  $a^2$  .

```
test =  $\frac{f[a] + g[b]}{f[a, b]}$ ;
test /. f[x_] → x^2
```

$$\frac{a^2 + g[b]}{f[a, b]}$$

Nell'esempio che segue, la variabile-pattern **f[x\_, y\_]** fa riferimento a qualsiasi espressione avente Head **f** ed in cui **f** sia funzione di **due variabili**, **x** ed **y** .

Pertanto **f[x\_, y\_]** combacia con  $f[a,b]$ .

La regola di sostituzione qui e'  $f[x_, y_] \rightarrow (x+y)^2$  .

Ne segue che  $f[a,b]$  e' sostituito da  $(a+b)^2$ .

```
test =  $\frac{f[a] + g[b]}{f[a, b]}$  ;
test /. f[x_, y_] → (x + y)^2
```

$$\frac{f[a] + g[b]}{(a + b)^2}$$

```
test =  $\frac{f[a] + g[b]}{f[a, b]}$  ;
test /. f[x_, y_] → x^2
(* Cerca qualsiasi espressione con Head f funzione di due variabili.
   Combacia con f[a,b] al denominatore di test .
   La regola di sostituzione e' f[x_,y_] → x^2.
   Pertanto f[a,b] viene sostituito con a^2 *)
```

$$\frac{f[a] + g[b]}{a^2}$$

Notiamo che in nessuno dei casi qui sopra la sottoespressione  $g[b]$  e' stata coinvolta, dato che la sua Head non combacia con la variabile-pattern.

⌘ Si deve sempre tenere a mente che i pattern combaciano con espressioni basate sulle **forma interna** (FullForm) delle espressioni stesse.

Non considerare la FullForm potrebbe generare confusione, qualora si cerchi di modificare una espressione la cui forma interna sia diversa da quella che vediamo sullo schermo. Consideriamo l'esempio che segue:

```
Clear[test, x, y];
test =  $\frac{x}{\text{Exp}[y]}$  ;
test // OutputForm
```

$$\frac{x}{y}$$

Usiamo la regola  $\text{Exp}[t_] \rightarrow t$

Se l'intento fosse stato quello di ottenere  $\frac{x}{y}$ , resteremmo sorpresi dal risultato della sostituzione:

```
test =  $\frac{x}{\text{Exp}[y]}$  ;
test /. Exp[t_] → t
```

$-x y$

Capiamo il motivo del risultato della sostituzione, esaminando la forma interna (FullForm) di **test** pattern :

```
Map[FullForm, {test, Exp[t_]}]
```

```
{Times[Power[E, Times[-1, y]], x], Power[E, Pattern[t, Blank[]]]}
```

```
{Times[Power[E, Times[-1, y]], x], Power[E, Pattern[t, Blank[]]]}
```

```
{e-y x, et-}
```

La sostituzione e' fatta col pattern **Power[E, Pattern[t, Blank[]]]**  
 seguendo la regola **Power[E, Pattern[t, Blank[]]] → t**  
 Qui il match e'  
**Power[E, Times[-1, y]] → Times[-1, y]**  
 ossia **E<sup>-y</sup> → -y** ovvero **Exp[-y] → -y**

```
{Power[E, Times[-1, y]], Times[-1, y]}
```

```
{e-y, -y}
```

Dunque otteniamo:

```
test =  $\frac{x}{\text{Exp}[y]}$ 
test /. Exp[t_] → t
(* Times e' n-aria *)
(* test/.Exp[t_]→t//FullForm *)
```

```
e-y x
```

```
-x y
```

Se l'intento fosse stato quello di ottenere  $\frac{x}{y}$  , avremmo potuto definire il pattern seguente:

```
test /. Exp[t_] → -1/t
```

```
 $\frac{x}{y}$ 
```

Alternativa. Per ottenere  $\frac{x}{y}$  , avremmo potuto definire il pattern seguente:

```
test /. 1/Exp[t_] → 1/t
```

$$\frac{x}{y}$$

## Programmazione basata su Regole

### ■ 6.1. Pattern \*

#### □ 6.1.2 De-strutturare

Un pattern puo' essere costruito da qualsiasi espressione semplicemente sostituendo Blank con varie sottoespressioni.

Come gia' detto, il termine Blank viene usato allo scopo di dare l'idea di << riempire dei vuoti >>.

A qualsiasi Blank puo' essere dato un nome, in modo da utilizzarlo come variabile—pattern.

```
expr3 = f[a] + g[b];
expr3 // FullForm
```

```
Plus[f[a], g[b]]
```

La variabile—pattern **x** combacia con la Head di qualsiasi espressione avente **una singola parte** (e restituisce la Head stessa):

```
expr3 /. x_[] → x
```

```
f + g
```

Qui sotto, **x** combacia con la Head di qualsiasi espressione avente **esattamente due parti** (e restituisce la Head stessa):

In questo esempio particolare, **x** combacia con Plus[a, b]

```
expr3 /. x_[], _ → x
```

```
Plus
```

L'esempio seguente illustra che e' possibile fare praticamente qualsiasi cosa, mediante de-strutturazione.

```
expr3 = f[a] + g[b];
expr3 /. x_[y_] → y[x]
```

```
a[f] + b[g]
```

```
a[f] + b[g]
```

Per modificare una sottoespressione risulta, in genere, semplice de-strutturare ed usare regole di sostituzione (si puo' usare anche MapAt, ma e' un diverso paradigma di programmazione).

Capire (visivamente) quello che una operazione di de-strutturazione sta facendo e' spesso piu' facile (che capire a quale parte di una espressione si riferisca una lunga sequenza di pedici).

C'e' una sola occasione in cui e' meglio usare pedici piuttosto che de-strutturare: quando una espressione contiene molte sotto-espressioni, ciascuna avente identica struttura & una sola di tali sotto-espressioni deve essere estratta o modificata.

Supponiamo di avere una lista di dati  $\{x, y\}$ , rappresentanti punti che vogliamo plottare in scala logaritmica (esiste la built-in LogPlot, ma supponiamo di voler scrivere noi una funzione equivalente).

Un modo (della programmazione funzionale) di trasformare i dati potrebbe essere come segue:

```
data = {{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}};
data // MatrixForm
```

$$\begin{pmatrix} x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \\ x_4 & y_4 \end{pmatrix}$$

Separiamo i valori x ed y

```
tdata = Transpose[data]
tdata // MatrixForm
```

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ y_1 & y_2 & y_3 & y_4 \end{pmatrix}$$

Usiamo **MapAt** per trasformare i valori **y** in scala logaritmica.

Prima ricordo come funziona MapAt:

? MapAt

Symbol

i

**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*, {{*i*<sub>1</sub>, *j*<sub>1</sub>, ...}, {*i*<sub>2</sub>, *j*<sub>2</sub>, ...}, ...}] applies *f* to parts of *expr* at several positions.

**MapAt**[*f*, *pos*] represents an operator form of **MapAt** that can be applied to an expression.

▼

```
(* Applico f alla posizione 2 della lista {a,b,c,d} *)
MapAt[f, {a, b, c, d}, 2]
(* Equivalento : MapAt[f,{a,b,c,d},{2}] *)

{a, f[b], c, d}
```

```
(* Applico f alle posizione 1 e 4 della lista {a,b,c,d} *)
MapAt[f, {a, b, c, d}, {1}, {4}]

{f[a], b, c, f[d]}
```

```
(* Applico f alle posizione 1 della parte 2 della lista {{a,b,c},{d,e}} *)
MapAt[f, {{a, b, c}, {d, e}}, {2, 1}]

{{a, b, c}, {f[d], e}}
```

Riprendiamo l'esempio con **tdata**.

Usiamo **MapAt** per trasformare i valori **y** (contenuti nella Parte 2 della matrice trasposta **tdata**) in scala logaritmica.

Questa operazione sfrutta il fatto che Log e' Listable (cfr. paragrafo 3.3).

```
tdata
(* Applichiamo Log alla posizione 2 della lista tdata *)
mtdata = MapAt[Log, tdata, 2]
mtdata // MatrixForm

{{x1, x2, x3, x4}, {y1, y2, y3, y4}}
```

```
 {{x1, x2, x3, x4}, {Log[y1], Log[y2], Log[y3], Log[y4]}}
```

$$\begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ \text{Log}[y_1] & \text{Log}[y_2] & \text{Log}[y_3] & \text{Log}[y_4] \end{pmatrix}$$

Ricombino i valori **x** ed i valori **Log[y]**

```
tmtdata = Transpose[mtdata];
tmtdata
tmtdata // MatrixForm
```

$$\begin{pmatrix} x_1 & \text{Log}[y_1] \\ x_2 & \text{Log}[y_2] \\ x_3 & \text{Log}[y_3] \\ x_4 & \text{Log}[y_4] \end{pmatrix}$$

```
{x1, Log[y1]}, {x2, Log[y2]}, {x3, Log[y3]}, {x4, Log[y4]}
```

$$\begin{pmatrix} x_1 & \text{Log}[y_1] \\ x_2 & \text{Log}[y_2] \\ x_3 & \text{Log}[y_3] \\ x_4 & \text{Log}[y_4] \end{pmatrix}$$

Il procedimento qui sopra puo' essere eseguito piu' elegantemente con i pattern (pattern matching)

```
data = {{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}};
datafast = data /. {x_, y_} → {x, Log[y]}
datafast // MatrixForm
data
```

```
{x1, Log[y1]}, {x2, Log[y2]}, {x3, Log[y3]}, {x4, Log[y4]}
```

$$\begin{pmatrix} x_1 & \text{Log}[y_1] \\ x_2 & \text{Log}[y_2] \\ x_3 & \text{Log}[y_3] \\ x_4 & \text{Log}[y_4] \end{pmatrix}$$

```
{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}
```

#### ▫ Esercizio 1 pagina 144

Che succede nei casi seguenti?

```
(* NO *)
dataDue = {{x1, y1}, {x2, y2}}
(* La lista dataDue ha esattamente 2 sottoparti .
   x_ intercetta la parte 1 di dataDueBis;
   y_ intercetta la parte 2 di dataDue *)
dataDue /. {x_, y_} → {x, Log[y]}
```

```
{x1, y1}, {x2, y2}
```

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

```
(* NO *)
dataDueBis = {{x1, y1, z1}, {x2, y2, z2}}
(* La lista dataDueBis ha esattamente 2 sottoparti .
  x_ intercetta la parte 1 di dataDueBis;
  y_ intercetta la parte 2 di dataDueBis *)
dataDueBis /. {x_, y_} → {x, Log[y]}

{{x1, y1, z1}, {x2, y2, z2}}
```

```
 {{x1, y1, z1}, {Log[x2], Log[y2], Log[z2]}}
```

```
(* OK *)
dataUno = {{x1, y1}}
dataUno /. {x_, y_} → {x, Log[y]}

{{x1, y1}}
```

```
 {{x1, Log[y1]}}
```

```
(* OK *)
dataTre = {{x1, y1}, {x2, y2}, {x3, y3}}
dataTre /. {x_, y_} → {x, Log[y]}

{{x1, y1}, {x2, y2}, {x3, y3}}
```

```
 {{x1, Log[y1]}, {x2, Log[y2]}, {x3, Log[y3]}}
```

# Basic Graphics Objects

## Circle

In *Mathematica*, **Circle[ ]** represents a circle, centered at {0,0} and of unitary radius.  
To display it, use the built-in **Graphics**.

To see how to specify position and size of a circle, read the Help Page of **Circle**.  
For now, we just deal with the unit circle, which does not need any input.

```
Graphics[Circle[], ImageSize → Tiny]
```

## Disk represents a filled - in disk :

```
full = Graphics[ Disk[]];
sector = Graphics[ Disk[ {0, 0}, 1 , {Pi/4, Pi/2} ] ];
(* Disk[]      is Disk[{0,0}, 1] *)
(* Disk[{x,y}]  is Disk[{x,y}, 1] *)(* Disk[ {x,y}, {rx,ry}, {θ1,θ2} ] is the filled
region{ { x + ρ*rx*cos(θ), y + ρ*ry*sin(θ)}   with θ1≤θ≤θ2 && 0≤ρ≤1 } *)
(* θ1, θ2 measured in radians counter-clock-wise from the positive X-axis *)
GraphicsRow[{full, sector}, Spacings → 50, ImageSize → Small]
```

Use the graphics transparency directive **Opacity** :

```
g1 = Graphics[{
  Opacity[0.3],
  Disk[{0, 0}, {2, 1}], Disk[{2, 2}, {1, 1}]
};

g2 = Graphics[
  Style[
    {Disk[{0, 0}, {2, 1}], Disk[{2, 2}, {1, 1}]},
    Opacity[0.3]
  ]
];
```

```
GraphicsRow[{g1, g2}, Spacings → 50, ImageSize → Small]
(* FullForm[g1]
 FullForm[g2] *)
```

Use the graphics directive **EdgeForm** :

```

Graphics[
  EdgeForm[{Thick, Blue}],
  Opacity[0.9],
  Red,
  Disk[{0, 0}, {2, 1}]
}, ImageSize → Tiny]
(* Red is equivalent to RGBColor[1,0,0]  * *)
(* RGBColor  is a graphics directive *)
(* ImageSize  is an Option *)

```

Create illusory contours :

```

Graphics[
  Disk[{0, 0}, 2, {Pi/3 , 2 Pi }],
  Disk[{6, 0}, 2, {-Pi , 2 Pi/3}],
  Disk[{3, 5}, 2, {4 Pi/3, -Pi/3}]
}, ImageSize → Tiny]

```

Make an oval pie-chart :

```

SeedRandom[1];
data = Reverse[Sort[RandomReal[1, 5]]];
(* RandomReal[1,5] gives a random real in [1,5] *)
(* data is {0.817389,0.789526,0.241361,0.187803,0.11142} *)

```

```

pie[data_] := Module[
  {t = 0, xc = 0, yc = 0, rx = 2, ry = 1, len, sum, dataNorm, paramopac = 0.8},
  (* t: angle spanning sectors in the pie-chart *)
  (* xc,yc: center of the pie-chart *)
  (* rx,ry: xradius and yradius of oval the pie-chart *)
  (* paramopac: parameter for Opacity used later, in Graphics *)

  len = Length[data];
  sum = Total[data];
  (* dataNorm are data normalized in [0,1] *)
  dataNorm = data/sum;

  (* Print the table of angle pairs {t_{k-1}, t_k}
   used to define sectors in the pie-chart *)
  Print[
    Table[{t, t += 2 Pi dataNorm[[k]]}, {k, len}]
    (* Part[expr, k] or expr[[k]] *)
    (* AddTo: x += X pre-incremental updating x=x+X *)
  ];
]

(* Define and render sectors forming the oval pie-chart *)
Graphics[
  Table[
    {
      (* k/len is in [0,1] *)
      Hue[k/len],
      EdgeForm[Opacity[paramopac]],
      (* A sector is defined as Disk[{xc,yc}, {rx,ry}, {θ_{k-1},θ_k}] *)
      Disk[ {xc, yc}, {rx, ry}, {t, t += 2 Pi dataNorm[[k]]} ]
    },
    {k, len}], (* end Table *)
  ImageSize → Tiny](* end Graphics *)
](* end Module *)
]

pie[data]

```

### NOTE 1. Operators of (pre) increment / decrement

A variable to be incremented must have an initial value,  
 i.e., x can be a symbol or other expression with an existing value.

```
Clear[k]; k = 3; {k, AddTo[k, 2], k} (* OK *)
Clear[k, d]; k = 3; {k, AddTo[k, d], k} (* OK *)
Clear[k]; {k, AddTo[k, 2], k} (* NO *)
```

**AddTo[x, dx]** or  $x += dx$  is equivalent to updating  $x = x + dx$   
 i.e.  $dx$  is added to  $x$ , the new  $x$  is used (e.g. in a test), the **new**  $x$  is **returned**  
 $\text{AddTo}[x, dx]$  is the generalization of  $\text{PreIncrement}[x]$  to any increment  $dx$

```
Clear[k]; k = 3; {k, AddTo[k, 1], k}
Clear[k]; k = 3; {k, PreIncrement[k], k}
Clear[k]; k = 3; {k, Increment[k], k}
```

**PreIncrement[x]** or  $++x$  equivalent to  $x += 1$  i.e.  $x = x + 1$

**1** is added to  $x$ , the new  $x$  is used, the **new**  $x$  is returned

**Increment[x]** or  $x++$  equivalent to  $x = x + 1$

**1** is added to  $x$ , the new  $x$  is used, the **old**  $x$  is returned

**Increment** and **PreIncrement** differ only in the **return value** of the operation :

```
Clear[a, b];
{a, b} = {1, 1};
{a++, ++b} (* {Increment[a], PreIncrement[b]} *)
{a, b}

(* SubtractFrom, PreDecrement, Decrement *)
```

**RegularPolygon[ n ]** gives a regular polygon with **n > 2** sides .

```
(* pentagon *)
Graphics[RegularPolygon[5], ImageSize -> Tiny]
```

You can find a **list** of REGULAR POLYGONS NAMES , for instance, at  
<https://www.mathsisfun.com/geometry/polygons.html>

```
(* Hyperlink["https://www.mathsisfun.com/geometry/polygons.html"] *)
```

```
(* triangle or trigon, quadrilateral or tetragon,
pentagon, hexagon, heptagon, octagon *)
gr = Table[
  Graphics[RegularPolygon[n], ImageSize → Tiny],
  {n, 3, 8}]

GraphicsRow[gr]

(* Another way, using Map *)
rp = Table[RegularPolygon[n], {n, 3, 8}];
mgr = Map[Graphics, rp];
GraphicsRow[ mgr ];

(* Different sizes *)
gr3 = Table[
  Graphics[ RegularPolygon[3] , ImageSize → s],
  {s, {Tiny, Small}}]

GraphicsRow[ gr3]

(* One Table with two iterators *)
grs = Table[
  Graphics[RegularPolygon[n], ImageSize → s],
  {s, {Tiny, Small}},
  {n, 3, 8}
];
(* TableForm[grs] *)
GraphicsGrid[grs]

(* Another Table with the two iterators swapped *)
grs = Table[
  Graphics[RegularPolygon[n], ImageSize → s],
  {n, 3, 8},
  {s, {Tiny, Small}}
];
GraphicsGrid[grs];
```

**Style** works inside **Graphics**, so you can use it to give colors.

```
Graphics[  
{Orange, RegularPolygon[5]},  
ImageSize → Tiny]
```

```
Graphics[  
Style[RegularPolygon[5], Orange],  
ImageSize → Tiny]
```

Another way to specify a **Style** for graphics is to give graphical directives (like **Orange**) in a list, before the graphical object of interest

```

tria = RegularPolygon[3];
penta = RegularPolygon[5];
dodeca = RegularPolygon[12];

p1 = {Orange, penta};
g1 = Graphics[p1];

p2 = {tria, (* default color is Black *)
      Orange, Opacity[0.3], penta, dodeca};
g2 = Graphics[p2];

p3 = {dodeca,
      p1, (* {Orange,penta} *)
      Blue, tria};
g3 = Graphics[p3];

p4 = {tria,
      Orange,
      {Opacity[0.3], dodeca},
      penta};
g4 = Graphics[p4];

(* Flatten *)
p5 = Flatten[p3];
g5 = Graphics[p5];

GraphicsRow[{g1, g2, g3, g4, g5}, Spacings → 50]

(* In questo caso, in p3, usare List attorno a
   { Orange, penta } == List[RGBColor[1,0.5` ,0],RegularPolygon[5]],
   risulta superfluo *)
FullForm[p3]
FullForm[p5]

```

**Sphere, Cylinder , Cone** are 3D constructs, that can be rendered with **Graphics3D**.

You can rotate 3D graphics interactively to see different angles.

```

Graphics3D[
  Sphere[],
  ImageSize → Tiny]

```

- A **list** of `Graphics3D` directives (Cone and Cylinder).
- One `Graphics3D`, whose argument is a list of `graphics` objects (Sphere and Cylinder) .

```

g3d = {
    Graphics3D[ Cone[]],
    Graphics3D[ Cylinder[]]
};

(* ccl={ Cone[], Cylinder[] } ;
   g3d=Map[ Graphics3D, ccl ] ; *)
GraphicsRow[g3d, ImageSize → Small]

Graphics3D[
{ Sphere[{0, 0, 2}], Cylinder[] },
ImageSize → Tiny]

```

A yellow sphere; note that it is rendered like an actual 3D object, with lighting; if it were pure yellow, we would not see any 3D depth, and it would look like a 2D disk.

```

Graphics3D[
Style[Sphere[], Yellow],
ImageSize → Tiny]

```

## Have a look at the Help Page of **ImageSize**.

- For instance:

`ImageSize → width` is equivalent to `ImageSize → {width, Automatic}`, while `ImageSize → {Automatic, height}` determines image size from height.

- `ImageSize` is an option not only for `Graphics`, but also for objects such as `Slider`, `Button`, `Grid`, `Pane` (pannello), and for built-in like **Import / Export**.

```

rose = Import["ExampleData/rose.gif"];
tinyRose = Import["ExampleData/rose.gif", ImageSize → Tiny];
gr = GraphicsRow[{rose, " ", tinyRose}]

? ImageSize

SetDirectory[NotebookDirectory[]];
marzotto = Import["marzotto.png"];

```

## NOTE 2. Setting the working Directory

```

Directory[];
(* Gives the current working directory, returning its full name as a string *)
(* CloudDirectory[] gives a CloudObject representing
the current working directory used for cloud objects *)

```

```

NotebookDirectory[];
(* Gives the directory of the current evaluation notebook *)

SetDirectory[];
(* Sets the current working directory to your "home" directory.
   It is equivalent to SetDirectory[ $HomeDirectory ] *)
(* SetCloudDirectory[] sets the current working directory
   for cloud objects to $CloudRootDirectory *)

SetDirectory[ NotebookDirectory[] ];

$Path ;
(* gives the default list of directories to search
   in attempting to find an external file *)

```

---

## Vocabulary

Circle[ ]	specify a circle
Disk[ ]	specify a filled-in disk
RegularPolygon[n]	specify a regular polygon with n sides
Graphics[object]	display an object as graphics
Sphere[], Cylinder[], Cone[], ...	specify 3D geometric shapes
Graphics3D[object]	display an object as 3D graphics
Opacity	
EdgeForm	
Module	
ImageSize	

---

## Exercises

**8.1** Use RegularPolygon to draw a triangle.

**8.2** Make graphics of a red circle.

**8.3** Make a red octagon.

**8.4** Make a list whose elements are disks with Hues varying from 0 to 1 in steps of 0.1.

**8.5** Make a Column of a red and a green triangle (SetDelayed).

**8.6** Make a **list** giving the regular polygons with 5 through 8 sides, with each polygon being colored pink (SetDelayed with default valued variables).

**8.7** Make a graphic of a purple cylinder.

**8.8** Make a list of polygons with 8, 7, 6, ..., 3 sides, and colored with **RandomColor**; then show them all overlaid with the triangle on top (hint: apply **Graphics** to the list).

**+8.1** Make a list of 8 regular pentagons with random color.

**+8.2** Make a list formed by one 20-sided regular polygon and one disk.

**+8.3** Make a list of polygons with 10, 9, ..., 3 sides.

---

## More to Explore

Guide to Graphics in *Mathematica*:

(\* [Hyperlink\["pagina", " link a pagina "\]](#) \*)

<https://reference.wolfram.com/language/guide/SymbolicGraphicsLanguage.html>

## Programmazione basata su Regole

### ■ 6.1. Pattern \*

#### □ 6.1.3 Testare pattern

MatchQ e Cases sono due funzioni per testare un pattern e capire con quale tipo di espressione esso combacia.

Il predicato MatchQ esegue un test per vedere se un pattern combacia con una espressione:

```
expr = a + b + c;
expr // FullForm
MatchQ[expr, _Plus]
(* expr ha come head Plus ? Si' *)
Plus[a, b, c]
```

```
True
```

Cases **seleziona** tutte le espressioni in una lista che combaciano con un dato pattern.

Cases e' utile per il debugging dei nostri pattern.

```
exprLista = {a, a+b, a+a}
pattern = x_+y_;
Cases[exprLista, pattern]
(* Solo il secondo elemento Plus[a,b] di exprLista combacia col pattern *)
(* Il terzo elemento di exprLista e' Times[2,a] *)
{a, a+b, 2 a}
```

```
{a + b}
```

Uso FullForm per capire l'output di Cases

```
(* exprLista={a,a+b,a+a};    pattern=x_+y_ ;  *)
exprLista // FullForm
pattern // FullForm
List[a, Plus[a, b], Times[2, a]]
```

```
Plus[Pattern[x, Blank[]], Pattern[y, Blank[]]]
```

Un altro esempio.

```
listaH = {a, a+b, HoldForm[a+a]}
listaH // FullForm
Cases[listaH, x_+y_]
Cases[listaH, _[x_+y_]]
(* Il secondo elemento di listaH combacia col pattern x_+y_ *)
(* Il terzo elemento di listaH combacia col pattern _[x_+y_] *)

{a, a+b, a+a}
```

```
List[a, Plus[a, b], HoldForm[Plus[a, a]]]
```

```
{a + b}
```

```
{a + a}
```

Il secondo argomento di Cases puo' essere una regola (Rule): in questo caso, tale regola viene applicata a ciascuna delle espressioni combacianti (prima del return dalla Cases stessa).

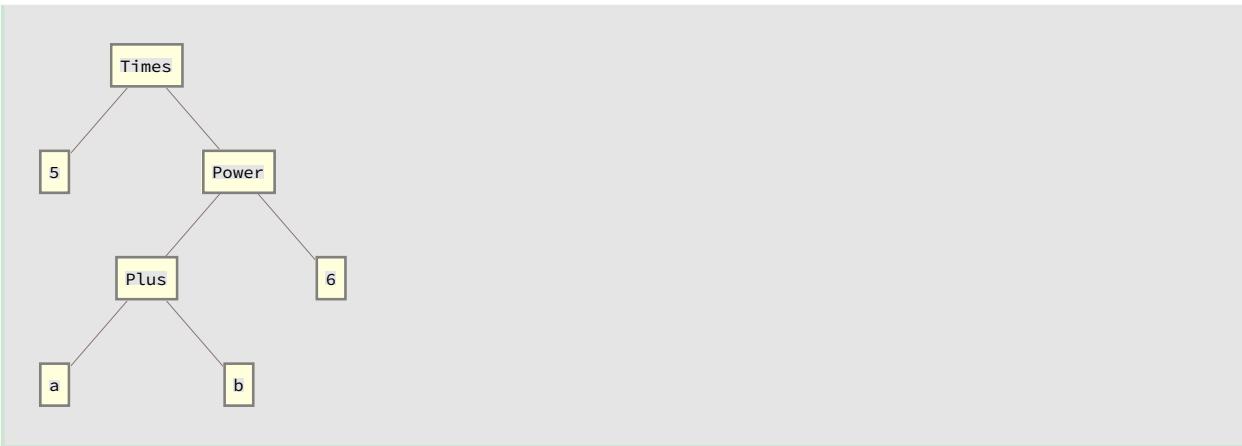
```
lista = {a, a+b, a+a};
Cases[lista, x_+y_]
Cases[lista, x_+y_ → y]
```

```
{a + b}
```

```
{b}
```

Nota. La Head del primo argomento di Cases non deve essere necessariamente List; inoltre, di default, Cases lavora SOLO al livello 1.

```
exprNonLista = 5 (a+b)^6;           -> posso usare la cases su exprNonLista e vedo che succede
TreeForm[exprNonLista, ImageSize → Small]
```



L' unico intero a livello 1 (livello di default per Cases) in esprNonLista e' 5.

Gli interi da livello 1 e fino al livello 2 in esprNonLista sono 5 e l' esponente 6

```
(* exprNonLista=5 (a+b)^6; *)
Cases[exprNonLista, _Integer]
Cases[exprNonLista, _Integer, 2]
```

{5}

{5, 6}

Cases ha dunque un terzo argomento opzionale per specificare il livello di applicazione.

**Infinity** specifica l'applicazione da livello 1 fino all'ultimo livello.

```
(* exprNonLista=5 (a+b)^6; *)
Cases[exprNonLista, _Integer, Infinity]
```

{5, 6}

Di default, Cases non considera le Head (a nessun livello).

Ma possiamo modificare tale scelta di default, specificando l'argomento opzionale Heads → True

```
(* exprNonLista=5 (a+b)^6; *)
Cases[exprNonLista, _Symbol, Infinity]
(* Gli unici Symbol in exprNonLista (escluse le Head) sono a,b *)
(* FullForm evidenzia tutti i Symbol in exprNonLista,
comprese le Head *)
exprNonLista // FullForm
Cases[exprNonLista, _Symbol, Infinity, Heads → True]
```

→ in questa chiamata, Cases andrà a controllare tutto (dal livello 0 fino all'ultimo)

```
{a, b}
```

```
Times[5, Power[Plus[a, b], 6]]
```

```
{Times, Power, Plus, a, b}
```

### Esercizio 1 pagina 145 (testo)

Con Select & MatchQ , implementare una propria versione di Cases (senza preoccuparsi della specifica del livello di applicazione).

Testare su : exprLista = {a , a + b , a + a , c + d} ; exprNonLista = 5 (a + b)^6 ;  
coi pattern : p1 = x\_ + y\_ ; p2 = \_Integer ; p3 = x\_\*y\_ ; p4 = \_Times ;

### **Es. 1 pag. 145 (soluzioni e discussione)**

### **Es. 2 pag. 145 (NON NEC)**

### Esercizio 3 pagina 145 (testo)

Creare un pattern che combaci con un simbolo (Symbol) elevato ad una potenza intera.

Tale pattern deve combaciare con espressioni quali **x^3** oppure **y^2** , mentre non deve combaciare con **x^(2/3)** e neppure con **(x+y)^2**.

Usare come test: testexpr = {x^3, y^2, x^(2/3), (x+y)^2} ; testexpr2 = {x^-1, y^3, x^y^3, x^2 + y^3, x^2 \* y^3};

### **Es. 3 pag. 145 (soluzioni e discussione)**

### Esercizio 4 pagina 145 (testo)

Il pattern dell' esercizio 3 precedente combacia con **x^1**?Se non combacia, spiegare perche' .

### **Es. 4 pag.145 (discussione)**

## Programmazione basata su Regole

### ■ 6.1. Pattern \*

#### □ 6.1.4 Il ruolo degli Attributi

Mathematica permette di de-strutturare.

Consideriamo l'esempio che segue.

L'espressione **a+b+c** ed il pattern **x\_+y\_** hanno strutture (e.g. Lunghezze) differenti; ciononostante, essi combaciano.

```
Clear[expr, pattern, rule, a, b, c];
expr = a + b + c;
pattern = x_ + y_;
(* /@ Map *)
(* Length /@ {expr, pattern} *)
Map[Length, {expr, pattern}]
(* Mappando Length sulla espressione a+b+c
   e sulla espressione x_+y_, vedo che
   la prima e' lunga 3 , mentre la seconda e' lunga 2.
   Nonostante questa differenza, MatchQ restituisce True *)
MatchQ[expr, pattern]
```

```
{3, 2}
```

```
True
```

Essi combaciano perche' il Kernel sa che Plus e' un operatore associativo i.e. **a+b+c == a+(b+c)**

```

rule = x_+y_ → {x, y};
{ a+b+c /. rule ,
  a+(b+c) /. rule ,
  (a+b)+c /. rule ,
  (a+c)+b /. rule }

(* per capire l'esito delle ReplaceAll qui sopra,
possiamo usare FullForm e/o la seguente catena di Equal *)
a + b + c == a + (b + c) == (a + b) + c == (a + c) + b

{{a, b + c}, {a, b + c}, {a, b + c}, {a, b + c}}

```

True

```

(* Per capire i risultati precedenti, usiamo FullForm *)
Map[
  FullForm,
  { a+b+c ,
    (a+b)+c ,
    Plus[Plus[a, b], c] }
]
{Plus[a, b, c], Plus[a, b, c], Plus[a, b, c]}

```

La conoscenza della associativita' di Plus e' codificata negli Attributi di Plus (come pure di Times)

```

Attributes[Plus]
Attributes[Times]
(* Attributes[Cases]*)

{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}

{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}

```

**Listable** significa che Plus viene automaticamente applicata (inserita, tracciata, threaded over) alle liste che appaiono come suoi argomenti.

In altre parole, se una funzione **f** e' listabile, allora **f[ {1,2,3} ]** restituisce **{ f[1], f[2], f[3] }**.

Nel caso di Plus : **Plus[ {a, b, c} , x ]** restituisce **{a+x, b+x, c+x}**

**Plus[ {a, b, c} , {x, y, z} ]** restituisce **{a+x, b+y, c+z}**

(in questo secondo caso, i due argomenti devono essere liste di lunghezza uguale)

**Flat** significa che Plus e' **associativa** (come visto negli esempi qui sopra).

**Onedentity** significa che **Plus[x]==x** i.e. che Plus agisce su un singolo argomento come se fosse la funzione **Identica** (i.e., su un singolo argomento, Plus agisce considerandolo come un suo punto fisso).

**Orderless** significa che Plus e' **commutativa** i.e.  $\text{Plus}[a, b] == \text{Plus}[b, a]$

**Protected** significa che non e' possibile definire nuove regole per Plus senza prima usare `Unprotect` (cfr. paragrafo 6.5 sulla re-scrittura di funzioni Built-in).

**? Listable**  
(\* una funzione f, listabile,  
viene "tracciata" su ogni elemento di una lista che sia argomento di f stessa \*)

Symbol

i

Listable is an attribute that can be assigned to a symbol f to indicate that the function f should automatically be threaded over lists that appear as its arguments.

▼

**? Flat**

(\* proprieta' associativa \*)

Symbol

i

Flat is an attribute that can be assigned to a symbol f to indicate that all expressions involving nested functions f should be flattened out. This property is accounted for in pattern matching.

▼

**? NumericFunction**

(\* Eg: NumericQ[Log[2]] returns True \*)

Symbol

NumericFunction is an attribute that can be assigned

to a symbol  $f$  to indicate that  $f[arg_0, arg_1, \dots]$  should be considered  
a numeric quantity whenever all the  $arg_i$  are numeric quantities.

**? OneIdentity**

(\* per Plus significa che Plus[x]==x \*)

Symbol

OneIdentity is an attribute that can be assigned to a symbol  $f$  to indicate that

$f[x]$ ,  $f[f[x]]$ , etc. are all equivalent to  $x$  for the purpose of pattern matching.

**? Orderless**

(\* proprietà commutativa \*)

Symbol

Orderless is an attribute that can be assigned to a symbol  $f$  to indicate that

the elements  $e_i$  in expressions of the form  $f[e_0, e_1, \dots]$  should automatically be  
sorted into canonical order. This property is accounted for in pattern matching.

**? Protected**

Symbol

Protected is an attribute that prevents

any values associated with a symbol from being modified.

```
a + b + c + d == (a + b) + (c + d)
```

```
True
```

L'attributo (di commutativita') Orderless fa si' che, nell'esempio qui sotto, il pattern `x_+ y_* z_` combaci con entrambe le sotto-espressionidi expr.

```

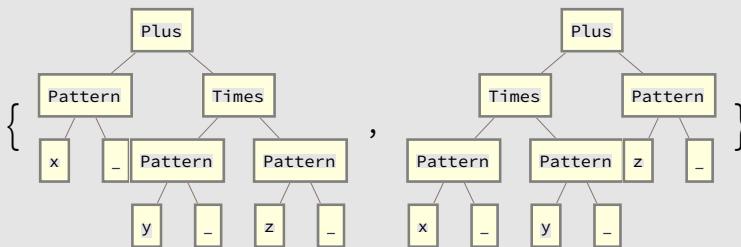
expr = {a+b*c, a*b+c}; expr // FullForm;
(* ordine alfanumerico && priorita' tra Times e Plus :
   Times viene eseguita prima di Plus ,
   quindi Times si trovera' ad un livello piu' verso le "foglie" ,
   mentre Plus e' a livello zero *)
pattern = x_ + y_* z_;
patternB = x_* y_ + z_;
{pattern // TreeForm, patternB // TreeForm}

(* uguaglianze *)
{pattern3 = x_ + z_* y_;
 pattern3 = y_* z_ + x_;
 pattern4 = z_* y_ + x_;
 pattern == pattern2 == pattern3 == pattern4,
 patternB2 = y_* x_ + z_;
 patternB3 = z_ + x_* y_;
 patternB4 = z_ + y_* x_;
 patternB == patternB2 == patternB3 == patternB4}

```

(\* pattern, patternB intercettano le sotto-espressioni di expr,  
nello stesso modo \*)

Cases[expr, pattern]  
Cases[expr, patternB]



{True, True}

{a+b c, a b+c}

{a+b c, a b+c}

La possibilita' di ottenere comportamenti quali quelli visti qui sopra permette di realizzare trasformazioni anche sofisticate, con poco sforzo.

Ad esempio, la regola che segue espande un prodotto (composto da un qualsiasi numero di termini).

```
(* definizione di una regola di espansione di un prodotto *)
expandrule = x_(y_+z_) → x*y+x*z;
```

La (applicazione della regola di) espansione puo' essere ripetuta, fintanto che continui ad esserci **un prodotto in cui uno dei fattori sia una somma di almeno due addendi** (come specificato dal pattern).

Quando nessun prodotto contiene piu' un termine fatto di almeno due addendi, l'espansione si ferma.

Applichiamo (ripetutamente) la regola "expandrule" alla espressione **a (b+c) (d+e+f)**.

```
expandrule = x_(y_+z_) → x*y+x*z;
(* Applicazione di expandrule *)
espr2 = a(b+c)(d+e+f);
espr2
```

$a(b+c)(d+e+f)$

Passo1. Il pattern **y\_ + z\_** combacia con **b + c**

Il pattern **x\_** combacia con qualsiasi altra cosa; in questo caso, esso combacia con **a (d+e+f)**.

Questo accade perche' **Times** ha Attributes: Flat (associativo) ed Orderless (commutativo).

```
(* y_ + z_ combacia con b+c mentre x_ combacia con a(d+e+f) *)
passo1 = espr2 /. expandrule
```

$a b (d + e + f) + a c (d + e + f)$

Passo2. Il pattern **y\_ + z\_** combacia con **d + (e+f)**

```
(* y_ + z_ combacia con d+(e+f)
mentre x_ combacia con a b nel prodotto a b (d+e+f)
e x_ combacia con a c nel prodotto a c (d+e+f) *)
passo2 = passo1 /. expandrule
```

$a b d + a c d + a b (e + f) + a c (e + f)$

Passo3. Il pattern **y\_ + z\_** combacia con **e+f**

```
(* y_ + z_ combacia con e+f
mentre x_ combacia con a b nel prodotto a b (e+f)
e       x_ combacia con a c nel prodotto a c (e+f) *)
passo3 = passo2 /. expandrule
```

```
a b d + a c d + a b e + a c e + a b f + a c f
```

```
(* RIPETO l'esercizio per avere tutti gli output , in cascata *)
expandrule = x_(y_+z_) → x y + x z ;
(* Applicazione ripetuta di expandrule *)
espr2 = a(b+c)(d+e+f); espr2
(* y_ + z_ combacia con b+c ; x_ combacia con a(d+e+f) *)
passo1 = espr2 /. expandrule
(* y_ + z_ combacia con d+(e+f);
x_ combacia con a b nel prodotto a b (d+e+f);
x_ combacia con a c nel prodotto a c (d+e+f) *)
passo2 = passo1 /. expandrule
(* y_ + z_ combacia con e+f ;
x_ combacia con a b nel prodotto a b (e+f) ;
x_ combacia con a c nel prodotto a c (e+f) *)
passo3 = passo2 /. expandrule
(* punto fisso *)
passo4 = passo3 /. expandrule;
passo4 == passo3
```

```
a (b + c) (d + e + f)
```

```
a b (d + e + f) + a c (d + e + f)
```

```
a b d + a c d + a b (e + f) + a c (e + f)
```

```
a b d + a c d + a b e + a c e + a b f + a c f
```

```
True
```

Dopo il Passo 3, qui sopra, non ci sono piu' **prodotti in cui almeno uno dei termini sia formato da almeno due addendi**.

Ripeto l'esercizio con una espressione piu' semplice (NON NEC)

Un modo piu' elegante di ottenere l'espansione ripetuta della Rule expandrule si puo' ottenere usando Fixed-Point.

**FixedPoint[ ]** applica una regola ad una espressione ripetutamente, fino a che l'espressione smette di cambiare (si arriva ad un Punto Fisso).

### ? FixedPoint

Symbol i

FixedPoint[f, expr] starts with *expr*, then applies *f* repeatedly until the result no longer changes.

▼

```
expandrule = x_(y_+z_) → x y + x z;
espr1 = a (d + e + f);
espr2 = a (b + c) (d + e + f);
(* definisco una funzione pura
   Function[ #/.expandrule ]      ovvero      #/.expandrule &
ed
uso FixedPoint *)
```

```
{espr1,
 FixedPoint[      # /. expandrule & , espr1]}
(* FixedPoint[ Function[ #/.expandrule ] , espr1] *)
```

```
{espr2,
 FixedPoint[ # /. expandrule & , espr2]}
(* FixedPoint[ Function[ #/.expandrule ] , espr2] *)
```

```
{a (d + e + f), a d + a e + a f}
```

```
{a (b + c) (d + e + f), a b d + a c d + a b e + a c e + a b f + a c f}
```

Questo tipo di procedimento ripetuto e' cosi' comune che e' stata scritta una funzione Built-in apposita per implementarlo.

Tale funzione e' la **ReplaceRepeated[]**, indicata anche con la forma speciale di input `//`.

```
expandrule = x_(y_+z_) → x y + x z;  
espr1 = a(d+e+f);  
espr2 = a(b+c)(d+e+f);  
  
{espr1,  
 espr1 // . expandrule}  
(* ReplaceRepeated[espr1,expandrule] *)
```

```
{espr2,  
 espr2 // . expandrule}  
(* ReplaceRepeated[espr2,expandrule] *)
```

```
{a (d + e + f), a d + a e + a f}
```

```
{a (b + c) (d + e + f), a b d + a c d + a b e + a c e + a b f + a c f}
```

### ? ReplaceRepeated

Symbol



*expr // . rules* repeatedly performs replacements until *expr* no longer changes.

`ReplaceRepeated[rules]` represents an operator

form of `ReplaceRepeated` that can be applied to an expression.



## Programmazione basata su Regole

### ■ 6.1. Pattern \*

#### □ 6.1.5 Funzioni che usano pattern

Abbiamo già usato alcune **funzioni che accettano pattern come argomenti** (anche se non sapevamo ancora che fossero pattern).

Il secondo argomento delle funzioni **Count** e **Position** è a tutti gli effetti un pattern.

#### ? Count

Symbol

i

Count[*list, pattern*] gives the number of elements in *list* that match *pattern*.

Count[*expr, pattern, levelspec*] gives the total number of subexpressions

matching *pattern* that appear at the levels in *expr* specified by *levelspect*.

Count[*pattern*] represents an operator form of Count that can be applied to an expression.

▼

Consideriamo il seguente esempio:

```
Clear[x, y]
expr = {a, a+b, a+b+c, a+a};
patt = x_ + y_;
(* Cases[] estrae le espressioni che combaciano col pattern *)
Cases[expr, patt]
(* Count[] conta quante espressioni combaciano col pattern *)
Count[expr, patt]
(* Position[] individua la posizione (nella lista)
delle espressioni che combaciano col pattern *)
Position[expr, patt]
```

{a + b, a + b + c}

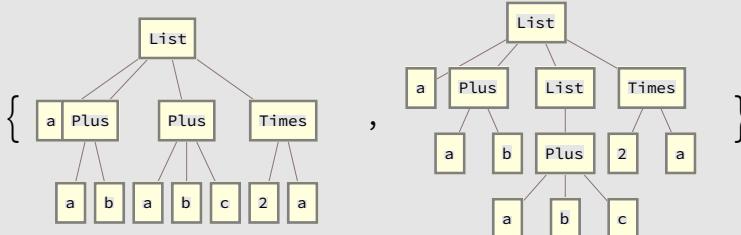
2

{{{2}, {3}}}

Come molte altre funzioni che accettano pattern, alle funzioni Count e Position può essere inoltre specificato

un livello, per circostanziare la loro ricerca.

```
expr = {a, a+b, a+b+c, a+a};
expr2 = {a, a+b, {a+b+c}, a+a};
{TreeForm[expr, ImageSize → Small], TreeForm[expr2, ImageSize → Small]}
```



```
patt = x_+y_;
expr2 = {a, a+b, {a+b+c}, a+a};
(* Individua i Match; il default e' il livello 1 *)
{Cases[expr2, patt],
 Cases[expr2, patt] == Cases[expr2, patt, 1]}
(* Individua i Match dal livello 1 fino al livello 2 *)
Cases[expr2, patt, 2]
{{a+b}, True}
```

```
{a+b, a+b+c}
```

```
patt = x_+y_;
expr2 = {a, a+b, {a+b+c}, a+a};
(* Numero di Match: il default e' il livello 1 *)
{Count[expr2, patt], TreeForm[expr, ImageSize → Small] ×
 Count[expr2, patt] == Count[expr2, patt, 1]}
(* Numero di Match da livello 1 fino al livello 2 *)
Count[expr2, patt, 2]
```

```
{1, True}
```

```
2
```

```
TreeForm[expr, ImageSize → Small]
```

```

patt = x_ + y_;
expr2 = {a, a+b, {a+b+c}, a+a};
(* Posizione dei Match a livello 1 *)
Position[expr2, patt, 1]
(* Posizione dei Match da livello 1 fino al livello 2 *)
Position[expr2, patt, 2]
(* Il default e' da livello 0 fino ad Infinity, con Heads → True *)
Position[expr2, patt]

```

{2}

{2}, {3, 1}

{2}, {3, 1}

Esiste anche la funzione DeleteCases che restituisce il complemento dell'output della Cases.  
Come Cases, DeleteCases puo' operare su espressioni aventi una Head qualsiasi.  
DeleteCases, inoltre, accetta specifiche di livello (come argomento opzionale).

### ? DeleteCases

Symbol

i

DeleteCases[*expr*, *pattern*] removes all elements of *expr* that match *pattern*.

DeleteCases[*expr*, *pattern*, *levelspec*] removes all

parts of *expr* on levels specified by *levelspec* that match *pattern*.

DeleteCases[*expr*, *pattern*, *levelspec*, *n*] removes the first *n* parts of *expr* that match *pattern*.

DeleteCases[*pattern*] represents an operator

form of DeleteCases that can be applied to an expression.

▼

```

patt = x_ + y_;
expr = {a, a+b, a+b+c, a+a};
Cases[expr, patt]
DeleteCases[expr, patt]

```

{a+b, a+b+c}

$\{a, 2a\}$

Secondo Esempio di DeleteCases

```
patt = x_+y_;
expr2 = {a, a+b, {a+b+c}, a+a};
TreeForm[expr2, ImageSize → Small]
```

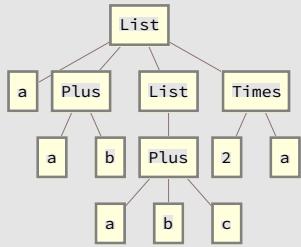
(\* DeleteCases[] individua i NON Match: il default e' il livello 1,

quindi il terzo elemento List[a+b+c] non e' un match \*)

```
{DeleteCases[expr2, patt], Cases[expr2, patt]}
```

(\* DeleteCases[] individua i NON Match da livello 1 fino al livello 2 \*)

```
{DeleteCases[expr2, patt, 2], Cases[expr2, patt, 2]}
```



$\{\{a, \{a+b+c\}, 2a\}, \{a+b\}\}$

$\{\{a, \emptyset, 2a\}, \{a+b, a+b+c\}\}$

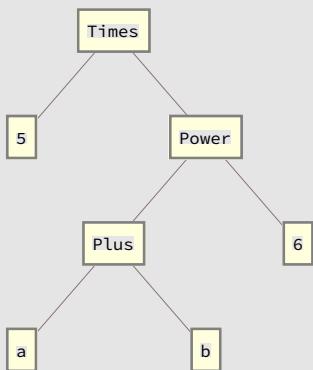
Terzo Esempio di DeleteCases

```

patt = _Integer ;
expr = 5 (a + b)^6; TreeForm[expr, ImageSize → Small]
(* Il valore di default per il livello e' {1} i.e. il solo livello 1 *)
{DeleteCases[expr, patt] ,
 Cases[expr, patt]}

{DeleteCases[expr, patt] ==
 DeleteCases[expr, patt, {1}] == DeleteCases[expr, patt, 1],
 Cases[expr, patt] == Cases[expr, patt, {1}] == Cases[expr, patt, 1]}

```



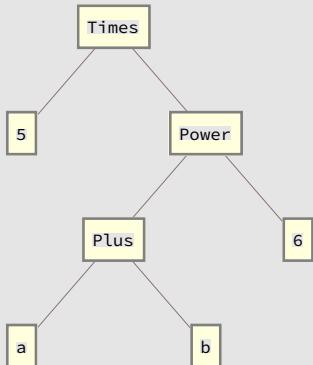
$\{(a + b)^6, \{5\}\}$

{True, True}

```

patt = _Integer ;
expr = 5 (a + b)^6; TreeForm[expr, ImageSize → Small]
(* Indicare il livello {n} significa
   specificare la ricerca solo sul livello n *)
{DeleteCases[expr, patt, {2}], Cases[expr, patt, {2}]}
(* Indicare il livello n significa
   specificare la ricerca da livello 1 fino ad n *)
{DeleteCases[expr, patt, 2], Cases[expr, patt, 2]}

```



$\{5 (a + b), \{6\}\}$

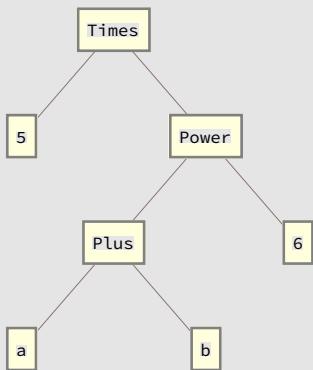
$\{a + b, \{5, 6\}\}$

```

patt = _Integer ;
expr = 5 (a + b)^6; TreeForm[expr, ImageSize → Small]
(* Indicare i livelli {m, n} significa
   specificare la ricerca da livello m ad n *)
{DeleteCases[ expr, patt, {1, 2}], Cases[ expr, patt, {1, 2}] }

(* Indicare il livello Infinity significa
   specificare la ricerca da livello 1 fino all'ultimo *)
{DeleteCases[ expr, patt, Infinity], Cases[ expr, patt, Infinity] }

```



{a + b, {5, 6}}

{a + b, {5, 6}}

# Interactive Manipulation

So far, we have been using *Mathematica* in a question-and-answer way: we type an input and obtain an output.

The built-in **Manipulate** serves to create an interface where we can continually manipulate one or more variables (parameters).

**Manipulate** works like **Table**: instead of producing a list of results, it gives a slider (by default) to interactively choose the parameters values.

## Manipulate (and Table)

```
tt[n_] := Table[Orange, n]; (* swatch *)
Table[      tt[n], {n, 1, 5, 1}]
Manipulate[tt[n], {n, 1, 5, 1}]
```

By default, Table assumes **start=step=1**.

Manipulate needs **start** to be specified.

And, if you omit the step size, Manipulate uses a non-integer step.

```
(* Importance of start and stepsize in Manipulate *)
cc[n_] := Column[{n, n^2, n^3}];
Table[cc[n], {n, 10}]
{Manipulate[cc[n], {n, 1, 10, 1}],
 Manipulate[cc[n], {n, 1, 10}]}]
```

## Manipulate and Charts

A bar chart that changes as you move the slider:

```
{Manipulate[
  BarChart[ {1, a, 4, 2*a, 4, 3*a, 1}, ImageSize → Small],
  {a, 0, 5}],
(* Inizializzo a "2" il parametro "a" nello slider *)
Manipulate[
  BarChart[ {1, a, 4, 2*a, 4, 3*a, 1}, ImageSize → Small],
  {{a, 2}, 0, 5}]}
```

A pie chart that changes as you move the slider:

```
Manipulate[
 Rotate[
 PieChart[{1, a, 4, 2*a, 4, 3*a, 1},
 ColorFunction → "BlueGreenYellow", ImageSize → Tiny],
 Pi / 2],
 {a, 0, 5}]
```

## Controls

**Manipulate** lets you set up any number of controls.

You just give the information for each variable in turn, one after another.

```
Manipulate[
 Graphics[
 Style[ RegularPolygon[n], Hue[h]], ImageSize → Tiny],
 (* n. of sides in the regular polygon *)
 {{n, 12, "n.sides"}, 5, 20, 1},
 (* color *)
 {{h, 0.5, "hue"}, 0, 1}
 ]
```

There are many ways to specify **Controls** for **Manipulate**.

A list of values generates a chooser or menu.

```
Manipulate[
 Graphics[
 Style[ RegularPolygon[5], color ], ImageSize → Tiny],
 {color, {Red, Yellow, Blue}}
 ]
```

If there are more than **5** choices, **Manipulate** sets up a drop-down menu:

```

Manipulate[
  Style[ number , color , size],
  {number, 1, 20, 1}, (* slider *)
  {color, {Blue, Red, Purple, Green, Black}}, (* chooser *)
  {size, Range[12, 96, 12]}(* drop-down menu *)
]

```

## More Examples

Examples of Manipulate with **Nest** (and with a Pure Function).

```

(* Subsuperscript is a built-in *)
{Subsuperscript[x, p, a], Subsuperscript[x, x, x]}
{xap, xxx}

myF[r_] := Subsuperscript[r, r, r];
myF[x]
Nest[ myF, x, 3]
(* Nest applies myF to "x", 3 times,
{x, myF[x], myF[myF[x]], myF[myF[myF[x]]]}, and yields the last result *)
NestList[myF, x, 3]
(* NestList gives the list of applications of myF to "x" *)

(* Nest[ f, expr, n] gives, as output, f applied n times to expr *)
(* Nest[f,expr,3]==f[f[f[expr]]] *)
(* NestList[f,expr,3]=={ expr, f[expr], f[f[expr]], f[f[f[expr]]] } *)

Manipulate[
Nest[myF, x, n], (* Nest applies myF to "x", n times *)
{n, 1, 5, 1}]

(* Version with a Pure Function *)
(* Subsuperscript[#, #2, #3] & *)
(* instead of myF[r_,s_,t_]:= Subsuperscript[r,s,t] *)
(* In this example, it was: myF[r_]:= Subsuperscript[r,r,r] *)
Manipulate[
(* Nest applies Subsuperscript[#, #2, #3]& to "x", n times *)
Nest[
  Subsuperscript[#, #2, #3]&,
  x, n],
{n, 1, 5, 1}]

```

## NOTE. Throw /Catch

**Nest** gives the last element of **NestList**

```
(* { (2^2), ((2^2)^2), ((2^2)^2)^2, ..... } *)
NestList[ #^2 &, 2, 5]
Nest[ #^2 &, 2, 5]
```

You can use **Throw /Catch** to exit from **Nest** before it is finished

```
(* Condition *)
Catch[
Nest[
If[ # > 10^2, Throw[##], #^2] &,
2,
5]
]
```

**Throw[val]** stops evaluation and returns **val** as the value of the nearest enclosing **Catch**.

```
(* Throw/Catch. Exit to the enclosing Catch as soon as Throw is evaluated:*)
Clear[a, b, c, d, e];
Print["Before Catch/Throw : {a,b,c,d,e} = ", {a, b, c, d, e}];
Catch[ a = 1; b = 2; Throw[c = 3]; d = 4; e = 5]
Print["After Catch/Throw : {a,b,c,d,e} = ", {a, b, c, d, e}];
```

The nearest enclosing **Catch** catches the **Throw**.

```
Clear[a, b, c, d, e];
Print["Before Catch/Throw : {a,b,c,d,e} = ", {a, b, c, d, e}];
Catch[ a = 1; b = 2; Throw[c = 3]; Throw[d = 4]; e = 5]
Print["After Catch/Throw : {a,b,c,d,e} = ", {a, b, c, d, e}];
```

**Catch[expr]** returns the argument of the first Throw that is evaluated.

**Catch[expr]** returns **expr** if there is no matching Throw.

```
Clear[a, b, c, d, e];
(* The inner Catch[ {a,Throw[b],c} ] returns b *)
(* The outer Catch[{b,d,e}] has no matching Throw, thus it returns {b,d,e} *)
Catch[
{ Catch[{a, Throw[b], c}], d, e }
]

Clear[a, b, c, d, e];
(* The inner Catch[{a, Throw[b], c}] returns b *)
(* The outer Catch[{b, Throw[d], e}] returns d *)
Catch[
{ Catch[{a, Throw[b], c}], Throw[d], e }
]
```

```

Clear[a, b, c, d, e];
(* The inner Catch[{a,b,c}] has no matching Throw, thus it returns {a,b,c} *)
(* The outer Catch[{{a,b,c}, Throw[d], e}]== Catch[{a,b,c, Throw[d], e}] returns d *)
Catch[
  { Catch[{a, b, c}], Throw[d], e }
]

```

Throw/Catch. Define a function that can “throw an exception” (see Help Page of Throw):

```

testF[x_] := If[x > 10, Throw["overflow"], x!];
Catch[testF[2] + testF[11]]
Catch[testF[2] + testF[8]]

```

Throw/Catch. Exit a loop when a criterion is satisfied:

```

Catch[
  Do[
    If[n! > 10^10, Throw[n]],
    {n, 20}]
]
{13! > 10^10, 14! > 10^10}

```

## Locator

```

Manipulate[
 Graphics[ Point[{p}] , PlotRange -> 2, ImageSize -> Tiny],
 {{p, {0, 0}}, Locator}

Manipulate[
 Graphics[{Red, Polygon[pt]}, PlotRange -> 2, ImageSize -> Tiny ],
 {
  {pt, {{0, 0}, {1, 0}, {1, 1}, {0, 1}}},
  Locator
 }]

```

With Windows, Mac, Linux, but possibly not in Cloud (?) :

LocatorAutoCreate → True :

Add locators with Ctrl Alt (clik)

Delete locators with Ctrl Alt (clik)

```

Manipulate[
 Graphics[{Line[u]}, PlotRange -> 2, ImageSize -> Tiny],
 {{u, {{-1, -1}, {1, 1}}}, 
  Locator,
  LocatorAutoCreate -> True}]

```

(\* Cntl Shift E per aprire e richiudere una cella e vederne il contenuto \*)

LocatorAutoCreate→All :  
 Add locators with with any mouse click  
 Delete locators with Ctrl Alt (clik)

```
myMan = Manipulate[
  Graphics[Point[locs], PlotRange → 2, ImageSize → Tiny],
  {{locs, {{0, 0}}}, 
   Locator,
   LocatorAutoCreate → All},
  SaveDefinitions → True]
```

## NOTE . **SaveDefinitions**

**SaveDefinitions** is an Option of Manipulate (and related functions) that specifies whether current definitions (relevant for the evaluation of the expression being manipulated) should automatically be saved.

```
f[x_] := x;
mf = Manipulate[f[x], {x, 0, 1}]
g[x_] := x;
mg = Manipulate[g[x], {x, 0, 1}, SaveDefinitions → True]
Warning! Saved variable definitions are effectively treated as global .
? g
```

## Further Examples

```
Manipulate[
 Plot3D[ Sin[a] Cos[b h], {a, 0, Pi}, {b, 0, Pi},
  ViewPoint → {1, 1, 0},
  ImageSize → Tiny],
 {h, 0, Pi}]

(* Manipulate[
 Plot3D[ Sin[a] Cos[b ], {a,0,Pi},{b,0,Pi},
  ViewPoint→{1,h,0},
  ImageSize→Tiny],
 {h,-1,1}] *) 

Manipulate[
 Plot[Sin[x (1 + x)], {x, 0, 6}, ImageSize → is],
 {is, 100, 200, 10}]
```

```
SetDirectory[NotebookDirectory[]];
rose = Import["ExampleData/rose.gif"];
Manipulate[Magnify[rose, zf], {zf, 0.1, 1, 0.1}]
```

## DynamicImage

```
DynamicImage[rose]
(* Creates a viewer with controls for zooming, panning,
and scrolling, useful for examining details of large images*)
DynamicImage[rose, ZoomFactor -> 3]
(* NO: Manipulate[DynamicImage[rose,ZoomFactor->zf],{zf,0.1,1,0.1}] *)
```

---

## Vocabulary

Manipulate[anything,{n,0,10,1}] manipulate anything with n varying in discrete steps of 1  
 Manipulate[anything,{x,0,10}] manipulate anything with x varying continuously  
 Locator  
 SaveDefinition  
 Nest  
 Throw , Catch

---

## Exercises

- 9.1** Make a Manipulate to show Range[n] with n varying from 0 to 100.
- 9.2** Make a Manipulate to ListPlot **Integers** up to **n**, where **n** can range from 5 to 50.
- 9.3** Make a Manipulate to show a Column of 1 to 10 copies of **x** (**ConstantArray**) .
- 9.4** Make a Manipulate to show a Disk with varying Hue.
- 9.5** Make a Manipulate to show a Disk with Red, Green, Blue color components, **each** varying from 0 to 1.
- 9.6** Make a Manipulate to show Digit sequences of 4-digit **Integers** (from 1000 to 9999).
- 9.7** Make a Manipulate to create a list of **n** Hues (equally spaced in [0,1]), with **n** varying between 5 and 15.

**9.8** Make a Manipulate that shows a list of  $n$  hexagons, with **Integer**  $n$  varying from 1 to 5, and with variable Hue (from 0 to 1).

**9.9** Make a Manipulate that shows a regular polygon having  $n$  sides, with  $n$  varying from 5 to 20, in Red or Blue or Yellow (or Green).

**9.10** Make a Manipulate that shows a PieChart divided in  $n$  equal segments, with  $n$  varying from 1 to 10.

**9.11** Make a Manipulate that gives a BarChart of the digits in 3-digit **Integers** (from 100 to 999).

**9.12** Make a Manipulate that shows  $n$  Random colors (swatches), where  $n$  ranges from 1 to 20.

**9.13** Make a Manipulate to display a Column of integer powers, with base from 1 to 15 and exponent from 1 to 9.

**9.14** Make a Manipulate of a NumberLinePlot of values of  $x^n$  (for Integer  $x$  from 1 to 10), with non-integer  $n$  varying from 0 to 3.

**9.15** Make a Manipulate to show a Sphere that can vary in color from Green to Red.

**+9.1** Make a Manipulate to ListPlot numbers  $n^p$ , with Integer  $n$  from 1 to 100, and  $p$  that can vary between -1 and +1.

**+9.2** Make a Manipulate to display 1000 at sizes between 9 and 40.

**+9.3** Make a Manipulate to show a BarChart with 4 bars, each with a height that can be between 0 and 10.

---

## Q & A

## Tech Notes

## More to Explore

- The [Wolfram Demonstrations Project](#): more than 11000 interactive Demonstrations created with Manipulate.

### 3.1 F is not closed w.r.t. operations

An operation between two finite numbers may yield a result that is not a finite number.

**Example 3.1** Consider  $x, y \in F(10, 2, p_{min}, p_{max})$ .

$$\begin{aligned} x &= 1.1 \cdot 10^0, & y &= 1.1 \cdot 10^{-2}, \\ x + y &= 1.1 \cdot 10^0 + 0.011 \cdot 10^0 = 1.111 \cdot 10^0. \end{aligned}$$

Result of the addition  $x + y \notin F(10, 2, p_{min}, p_{max})$ .

## F not closed w.r.t. operations (*continued*)

$$x, y \in F(\beta, t, p_{min}, p_{max}) \not\Rightarrow x + y \in F(\beta, t, p_{min}, p_{max})$$

This may happen in any of the arithmetic operations  $+, -, \times, /$  and in  $\sqrt{\cdot}$ .

The problem then arises of approximating, with a finite number, the result of an arithmetic operation between two (or more) finite numbers.

For each arithmetic operation  $+$ , it is necessary to define an equivalent finite (floating point) operation  $\oplus$ , in such a way that the result  $x \oplus y$  is a finite number that is a *good* approximation for  $x + y$ .

## 3.2 Aims of the IEEE Standard

Define the rules for:

- finite representation, i.e. how to correctly carrying out chopping / rounding on  $\alpha$  to obtain  $fl(\alpha)$ ;
- treating exceptional situations (exceptions);
- conversion of format;
- finite arithmetics, i.e. how to i.e. how to correctly carrying out floating point operation and their chopping / rounding.

### 3.3 Rules of the IEEE Standard

→ The operations  $\oplus, \ominus, \otimes, \oslash$  must be defined in such a way that  $\forall x, y$  finite numbers, it holds:

$$\begin{aligned} x \oplus y &= fl(x + y) \\ x \ominus y &= fl(x - y) \\ x \otimes y &= fl(x \times y) \\ x \oslash y &= fl(x/y) \end{aligned} \tag{3.1}$$

In other words, the finite number  $x \oplus y$ , resulting from an operation in finite arithmetics, must coincide with the chopping / rounding (carried out according to the rules for finite representation) of the real number  $x + y$ , computed in exact arithmetics.

### 3.4 One Machine Operation

An operation carried out in finite arithmetics is also called *finite operation* or *machine operation*.

From Corollary 2.1 we have that  $\forall x, y \in F(\beta, t, p_{min}, p_{max})$

$$\begin{aligned} fl(x + y) &= (x + y)(1 + \delta), & |\delta| \leq u \\ fl(x - y) &= (x - y)(1 + \delta), & |\delta| \leq u \\ fl(x \times y) &= (x \times y)(1 + \delta), & |\delta| \leq u \\ fl(x / y) &= (x / y)(1 + \delta), & |\delta| \leq u \end{aligned} \tag{3.2}$$

By considering (3.1) and (3.2), it follows that  $\forall x, y \in F(\beta, t, p_{min}, p_{max})$

$$\begin{aligned} x \oplus y &= (x + y)(1 + \delta), & |\delta| \leq u \\ x \ominus y &= (x - y)(1 + \delta), & |\delta| \leq u \\ x \otimes y &= (x \times y)(1 + \delta), & |\delta| \leq u \\ x \oslash y &= (x / y)(1 + \delta), & |\delta| \leq u \end{aligned} \tag{3.3}$$

### 3.5 Two or more Machine Operations

Machine Operations do **not** satisfy all the algebraic properties of operations carried out in the Field of Reals. For example, they do **not** hold:

- Associativity and Distributivity

$$(a + b) + c = a + (b + c)$$

$$(a \times b) \times c = a \times (b \times c)$$

$$a \times (b + c) = a \times b + a \times c$$

- Law of cancellation

$$a \times b = a \times c, a \neq 0 \Rightarrow b = c$$

- Law of simplification

$$a \times (b/a) = b \quad (\text{per } a \neq 0)$$

- Zero-product law

$$a \times b = 0 \implies a = 0 \text{ or } b = 0$$

### 3.6 Associativity doesn't hold in Finite Maths

→ The result of a sequence of finite operations may **not** coincide with the exact result and it **depends** on the order with which the finite operations are carried out.

In exact arithmetic:

$$x + y + w = x + (y + w) = (x + y) + w = (x + w) + y$$

In finite arithmetic?

**Example 3.2** Consider  $x, y, w \in F(10, 2, p_{min}, p_{max})$ . Let us use rounding.

$$x = 1.1 \cdot 10^0, \quad y = 1.3 \cdot 10^{-1}, \quad w = 1.4 \cdot 10^{-1}.$$

**Method 1.**

$$x \oplus y = fl(1.1 \cdot 10^0 + 0.13 \cdot 10^0) = fl(1.23 \cdot 10^0) = 1.2 \cdot 10^0$$

$$(x \oplus y) \oplus w = fl(1.2 \cdot 10^0 + 0.14 \cdot 10^0) = fl(1.34 \cdot 10^0) = 1.3 \cdot 10^0$$

By varying the order of execution of the operations, the result changes:

**Method 2.**

$$y \oplus w = fl(1.3 \cdot 10^{-1} + 1.4 \cdot 10^{-1}) = fl(2.7 \cdot 10^{-1}) = 2.7 \cdot 10^{-1}$$

$$x \oplus (y \oplus w) = fl(1.1 \cdot 10^0 + 0.27 \cdot 10^0) = fl(1.37 \cdot 10^0) = 1.4 \cdot 10^0$$

- In exact arithmetics:  $(x + y) + w = x + (y + w) = 1.37 \cdot 10^0$ .

Method 1 implies a relative error given by:

$$E_{rel} = \frac{|1.37 \cdot 10^0 - 1.3 \cdot 10^0|}{|1.37 \cdot 10^0|} = \frac{|7.0 \cdot 10^{-2}|}{|1.37 \cdot 10^0|} \approx 5.1 \cdot 10^{-2}$$

Method 2 implies a relative error given by:

$$E_{rel} = \frac{|1.37 \cdot 10^0 - 1.4 \cdot 10^0|}{|1.37 \cdot 10^0|} = \frac{|3.0 \cdot 10^{-2}|}{|1.37 \cdot 10^0|} \approx 2.2 \cdot 10^{-2}$$

### 3.8 Zero-Product doesn't hold in Finite Maths

In exact arithmetic:  $a \times b = 0 \implies a = 0$  or  $b = 0$

In finite arithmetic?

**Example 3.4** Consider  $x, y \in F(10, 7, -8, 9)$ .

With chopping:  $u = \varepsilon_{\text{machine}} = 10^{1-7} = 1.0 \cdot 10^{-6}$ .

With rounding:  $u = \varepsilon_{\text{machine}}/2 = 5.0 \cdot 10^{-7}$ .

$$x = 2.0 \cdot 10^{-5}, \quad y = 1.0 \cdot 10^{-4}.$$

$$x \otimes y = fl(2.0 \cdot 10^{-5} \times 1.0 \cdot 10^{-4}) = fl(2.0 \cdot 10^{-9}) \quad \leftarrow \text{underflow}$$

Here:  $x > u, y > u,$

$fl(x \times y)$  underflow, smaller than  $u$  and approximated by zero.

In other words:  $fl(x \times y) = 0$  even though  $x \neq 0, y \neq 0$ .

## Consequences

If we have  $x, y, w \in F(10, 7, -8, 9)$

$$x = 2.0 \cdot 10^{-5}, \quad y = 1.0 \cdot 10^{-4}, \quad w = 3.0 \cdot 10^{-1},$$

and we need to compute

$$\frac{w}{x \times y} \quad \leftarrow \text{not computable, as } fl(x \times y) = 0.$$

We might instead use, for example:

$$\frac{w}{x} \times \frac{1}{y}$$

taking into account that

$$fl\left(\frac{w}{x \times y}\right) \neq fl\left(\frac{w}{x} \times \frac{1}{y}\right)$$

## Programmazione basata su Regole

### ■ 6.1. Pattern \*

Count e Position prendono un'espressione e un pattern e ci restituiscono il numero del match (quante volte il pattern combacia con l'espressione) e le posizioni di questo match.

#### ▫ 6.1.5 Funzioni che usano pattern

Abbiamo gia' usato alcune **funzioni che accettano pattern come argomenti** (anche se non sapevamo ancora che fossero pattern).

Il secondo argomento delle funzioni **Count** e **Position** e' a tutti gli effetti un pattern.

#### ? Count

Symbol



Count[list, pattern] gives the number of elements in list that match pattern.

Count[expr, pattern, levelspec] gives the total number of subexpressions

matching pattern that appear at the levels in expr specified by levelspec.

Count[pattern] represents an operator form of Count that can be applied to an expression.



Consideriamo il seguente esempio:

```
Clear[x, y]
expr = {a, a+b, a+b+c, a+a};
patt = x_ + y_;
(* Cases[] estrae le espressioni che combaciano col pattern *)
Cases[expr, patt]
(* Count[] conta quante espressioni combaciano col pattern *)
Count[expr, patt]
(* Position[] individua la posizione (nella lista)
delle espressioni che combaciano col pattern *)
Position[expr, patt]
```

```
{a + b, a + b + c}
```

```
2
```

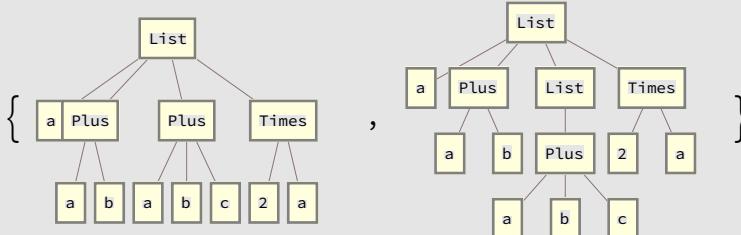
```
 {{2}, {3}}
```

Come molte altre funzioni che accettano pattern, alle funzioni Count e Position puo' essere inoltre specificato

un livello, per circostanziare la loro ricerca.

di default lavorano a livello 1

```
expr = {a, a+b, a+b+c, a+a};
expr2 = {a, a+b, {a+b+c}, a+a};
{TreeForm[expr, ImageSize → Small], TreeForm[expr2, ImageSize → Small]}
```



```
patt = x_ + y_;
expr2 = {a, a+b, {a+b+c}, a+a};
(* Individua i Match; il default e' il livello 1 *)
{Cases[expr2, patt],
 Cases[expr2, patt] == Cases[expr2, patt, 1]}
(* Individua i Match dal livello 1 fino al livello 2 *)
Cases[expr2, patt, 2]
{{a+b}, True}
```

{a+b, a+b+c}

```
patt = x_ + y_;
expr2 = {a, a+b, {a+b+c}, a+a};
(* Numero di Match: il default e' il livello 1 *)
{Count[expr2, patt], TreeForm[expr, ImageSize → Small] ×
 Count[expr2, patt] == Count[expr2, patt, 1]}
(* Numero di Match da livello 1 fino al livello 2 *)
Count[expr2, patt, 2]
```

{1, True}

2

TreeForm[expr, ImageSize → Small]

Position cerca i pattern in tutti i livelli, da 0 fino ad infinito

```
patt = x_ + y_;
expr2 = {a, a+b, {a+b+c}, a+a};
(* Posizione dei Match a livello 1 *)
Position[expr2, patt, 1]
(* Posizione dei Match da livello 1 fino al livello 2 *)
Position[expr2, patt, 2]
(* Il default e' da livello 0 fino ad Infinity, con Heads → True *)
Position[expr2, patt]

{{2}}
```

```
 {{2}, {3, 1}}
```

```
 {{2}, {3, 1}}
```

Esiste anche la funzione DeleteCases che restituisce il complemento dell'output della Cases.  
Come Cases, DeleteCases puo' operare su espressioni aventi una Head qualsiasi.  
DeleteCases, inoltre, accetta specifiche di livello (come argomento opzionale).

### ? DeleteCases

Symbol

i

**DeleteCases**[*expr*, *pattern*] removes all elements of *expr* that match *pattern*.  
**DeleteCases**[*expr*, *pattern*, *levelspec*] removes all  
parts of *expr* on levels specified by *levelspec* that match *pattern*.  
**DeleteCases**[*expr*, *pattern*, *levelspec*, *n*] removes the first *n* parts of *expr* that match *pattern*.  
**DeleteCases**[*pattern*] represents an operator  
form of DeleteCases that can be applied to an expression.

▼

```
patt = x_ + y_;
expr = {a, a+b, a+b+c, a+a};
Cases[expr, patt]
DeleteCases[expr, patt]

{a+b, a+b+c}
```

$\{a, 2a\}$

Secondo Esempio di DeleteCases

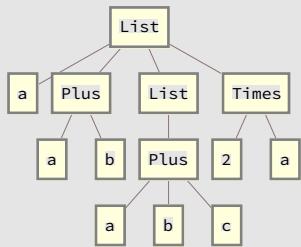
```
patt = x_+y_;
expr2 = {a, a+b, {a+b+c}, a+a};
TreeForm[expr2, ImageSize → Small]
```

(\* DeleteCases[] individua i NON Match: il default e' il livello 1,  
quindi il terzo elemento List[a+b+c] non e' un match \*)

$\{\text{DeleteCases}[expr2, patt], \text{Cases}[expr2, patt]\}$

(\* DeleteCases[] individua i NON Match da livello 1 fino al livello 2 \*)

$\{\text{DeleteCases}[expr2, patt, 2], \text{Cases}[expr2, patt, 2]\}$



$\{\{a, \{a+b+c\}, 2a\}, \{a+b\}\}$

$\{\{a, \emptyset, 2a\}, \{a+b, a+b+c\}\}$

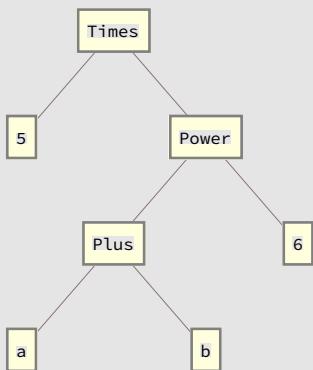
Terzo Esempio di DeleteCases

```

patt = _Integer ;
expr = 5 (a + b)^6; TreeForm[expr, ImageSize → Small]
(* Il valore di default per il livello e' {1} i.e. il solo livello 1 *)
{DeleteCases[expr, patt] ,
 Cases[expr, patt]}

{DeleteCases[expr, patt] ==
 DeleteCases[expr, patt, {1}] == DeleteCases[expr, patt, 1],
 Cases[expr, patt] == Cases[expr, patt, {1}] == Cases[expr, patt, 1]}

```



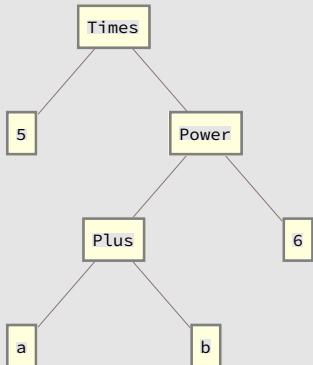
$\{(a + b)^6, \{5\}\}$

{True, True}

```

patt = _Integer ;
expr = 5 (a + b)^6; TreeForm[expr, ImageSize → Small]
(* Indicare il livello {n} significa
   specificare la ricerca solo sul livello n *)
{DeleteCases[expr, patt, {2}], Cases[expr, patt, {2}]}
(* Indicare il livello n significa
   specificare la ricerca da livello 1 fino ad n *)
{DeleteCases[expr, patt, 2], Cases[expr, patt, 2]}

```



$\{5 (a + b), \{6\}\}$

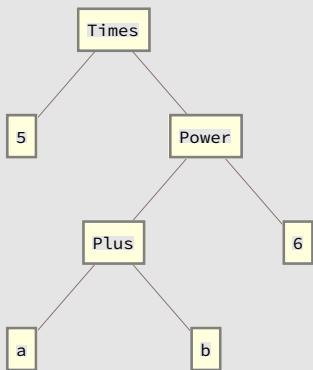
$\{a + b, \{5, 6\}\}$

```

patt = _Integer ;
expr = 5 (a + b)^6; TreeForm[expr, ImageSize → Small]
(* Indicare i livelli {m, n} significa
   specificare la ricerca da livello m ad n *)
{DeleteCases[ expr, patt, {1, 2}], Cases[ expr, patt, {1, 2}] }

(* Indicare il livello Infinity significa
   specificare la ricerca da livello 1 fino all'ultimo *)
{DeleteCases[ expr, patt, Infinity], Cases[ expr, patt, Infinity] }

```



{a + b, {5, 6}}

{a + b, {5, 6}}

## Programmazione basata su Regole

### ■ 6.2. Regole e Funzioni \*

In questo paragrafo vedremo che esiste una connessione stretta tra regole (Rule) e funzioni.  
Prima di proseguire, però, dobbiamo studiare un nuovo tipo di regola.

#### □ 6.2.2. Le definizioni di funzione sono regole : i DownValues

Quando definiamo una funzione **f** in **Mathematica** stiamo in effetti definendo una Regola (Rule).  
Le regole definite per **f** possono essere visualizzate (displayed) usando la Built-in **DownValues[f]**.  
**DownValues[ f ]** restituisce tutte le regole corrispondenti a definizione fatte per il simbolo **f**.

regola associata al simbolo della funzione

```
? DownValues
(* Possiamo specificare direttamente i DownValues per una f ,
con l'assegnazione DownValues[f] = list *)
(* La lista restituita dalla chiamata a
DownValues ha elementi nella forma HoldPattern[ lhs ] :> rhs *)
```

Symbol

i

DownValues[*f*] gives a list of transformation rules  
corresponding to all downvalues (values for *f*[...]) defined for the symbol *f*.

▼

```
? HoldPattern
```

Symbol

i

HoldPattern[*expr*] is equivalent to *expr* for  
pattern matching, but maintains *expr* in an unevaluated form.

▼

#### Esempio per capire i DownValues

Definiamo **f** con definizioni multiple e usiamo **/;** ossia **Condition[]**.

**f** è una funzione di  $x_$  e a seconda della sua grandezza fa cose diverse

```
Clear[f, g1, g2];
(*   Se x > -2 allora g1;
     Altrimenti e' x ≤ -2   && Se x < 2 (che e' Vero quando x ≤ -2) allora g2  *)
(* ⇒ se x > -2 allora g1 ;
   se x ≤ -2 allora g2  *)
f[x_] /; x > -2 := g1[x];           definiamo la funzione f per funzioni multiple
f[x_] /; x < 2 := g2[x];
DownValues[f] // TableForm

HoldPattern[f[x_] /; x > -2] → g1[x]
HoldPattern[f[x_] /; x < 2] → g2[x]
```

Le Regole definite (in corrispondenza delle assegnazioni) per un dato simbolo (funzione) **f** vengono interpretate nell'ordine in cui esse sono date (osserviamo che, qui, le due regole hanno la stessa specificità/generalità).

? f

Symbol
Global`f
Definitions
$f[x_] /; x > -2 := g1[x]$
$f[x_] /; x < 2 := g2[x]$
Full Name Global`f
^

Proseguiamo con l'esempio di **f** data con definizioni multiple (se  $x > -2$  allora **g1**; se  $x \leq -2$  allora **g2**). Studiamo **f** sulle ascisse { -3, -2, -1, 0, 1, 2, 3 }.

```
(* Studiamo f sulle ascisse {-3,-2,-1,0,1,2,3} *)
(* af e' un Array di Head "f" ,
con 7 componenti ed indice iniziale -3 *)
(* af serve come "stampa" di f,
per vederne la corrispondenza con g1,g2 *)
af = Array["f", 7, -3];
(* ag e' un Array di Head f : qui gli indici sono ascisse,
quindi f[x] diventa g1[x] oppure g2[x] *)
ag = Array[f, 7, -3];
tt = Table[{af[[i]], ag[[i]]}, {i, Length[af]}];
(* se x > -2 allora g1 ; se x ≤ -2 allora g2 *)
TableForm[tt]

f[-3]    g2[-3]
f[-2]    g2[-2]
f[-1]    g1[-1]
f[0]     g1[0]
f[1]     g1[1]
f[2]     g1[2]
f[3]     g1[3]
```

Possiamo usare DownValues e Reverse per invertire l'ordine in cui sono date le regole di definizione di f.

```
DownValues[f] // TableForm
```

```
HoldPattern[f[x_ /; x > -2]] :> g1[x]
HoldPattern[f[x_ /; x < 2]] :> g2[x]
```

```
DownValues[f] = Reverse[DownValues[f]];
DownValues[f] // TableForm
```

```
HoldPattern[f[x_ /; x < 2]] :> g2[x]
HoldPattern[f[x_ /; x > -2]] :> g1[x]
```

```
? f
(* Se x < 2 allora g2;
   Altrimenti e' x ≥ 2 && Se x ≥ -2 (che e' Vero quando x ≥ 2) allora g1 *)
(* ⇒ se x < 2 allora g2;
   se x ≥ 2 allora g1 *)
```

Symbol
Global`f
Definitions
f[x_ /; x < 2] := g2[x]
f[x_ /; x > -2] := g1[x]
Full Name Global`f
^

Ora **f** ha come definizioni multiple: se  $x < 2$  allora  $g2$ ; se  $x \geq 2$  allora  $g1$ .

Ri-studiamof sulle ascisse  $\{-3, -2, -1, 0, 1, 2, 3\}$ .

```
(* Ricrovo gli Array af, ag *)
af = Array["f", 7, -3];
ag = Array[f, 7, -3];
(* se x < 2 allora g2; se x ≥ 2 allora g1 *)
tt = Table[{af[[i]], ag[[i]]}, {i, Length[af]}];
TableForm[tt]

f[-3]    g2[-3]
f[-2]    g2[-2]
f[-1]    g2[-1]
f[0]     g2[0]
f[1]     g2[1]
f[2]     g1[2]
f[3]     g1[3]
```

## FINE ESEMPIO #

### Nota su HoldPattern

**HoldPattern** (che appare in DownValues) e' usato per "proteggere" le regole (Rule) dalla loro stessa definizione.

Altrimenti accadrebbe, ad esempio, quanto segue:

```

Clear[f];
f[x_] := x^2;
DownValues[f]
(* DownValues[f] restituisce HoldPattern[f[x_]] :> x^2 *)
(* Se non ci fosse HoldPattern e se venisse restituito f[x_] :> x^2 ,
allora verrebbe interpretata NON una assegnazione (ad f) differita,
bensì' una regola differita con output x^2 :> x^2 *)
f[x_] :> x^2

```

Osserviamo la differenza:

```

(* fa è definito con una Assegnazione differita *)
fa[x_] := x^2; fa[x] // TraditionalForm
fa[x] /. x → 0

```

$x^2$

0

```

(* fr è definito con una Regola differita *)
fr[x_] :> x^2; fr[x]
fr[x] /. x → 0

```

fr[x]

fr[0]

Sembra che al simbolo `gs` non venga associato nulla.

In effetti, è così, e lo possiamo vedere bene con `DownValues`:

```

Clear[fa, fr];
fa[x_] := x^2; DownValues[fa] // TraditionalForm
fr[x_] :> x^2;
DownValues[fr]

{HoldPattern[fa(x_)] :> x^2}

```

{}

Nota. Il fatto che ad `gs` non viene associato nulla viene segnalato anche dai colori (nero per `fa`, blu per `fr`)

```
? fa
? fr
```

Dato che (a differenza di quanto accade per `g`) ad `fr` non viene associato nulla, se generiamo una lista di ascisse (ad esempio, 10 numeri Interi Random tra -5 e 5) ed applichiamo `fa` ed `fr`, avremo il seguente comportamento:

```
(* fa[x_]:=x^2; *)
(* fr[x_]:=x^2; *)
SeedRandom[3];
trr = RandomInteger[{-5, 5}, 10]; trr
Map[fa, trr]
(* trr^2 == Map[fa,trr] *)
Map[fr, trr]
```

```
{2, 5, 3, -3, 3, -5, 4, 5, 4, -4}
```

```
{4, 25, 9, 9, 9, 25, 16, 25, 16, 16}
```

```
{fr[2], fr[5], fr[3], fr[-3], fr[3], fr[-5], fr[4], fr[5], fr[4], fr[-4]}
```

NOTA.

Per una descrizione di DownValues e UpValues, fare riferimento al tutorial  
"Associating Definitions with Different Symbols"

( <https://reference.wolfram.com/language/tutorial/TransformationRulesAndDefinitions.html> )

- **6.2.2. Le definizioni di funzione sono regole**
- **6.2.3. Definizioni multiple per lo stesso simbolo**
- **Esercizi**
- **6.2.4. Vantaggi della definizione di funzione basata su regole**
- **6.2.5. Rimuovere selettivamente le definizioni**
- **6.2.6. Programmazione "pura" basata su regole**
- **Esercizi**
- **6.3. Mattoni per costruire Pattern \***
- **6.4. Programmazione Dinamica \***
- **6.5. Andare oltre le Funzioni Built-In \***
- **6.6. Risorse addizionali \***

## Programmazione basata su Regole

### ■ 6.2. Regole e Funzioni \*

In questo paragrafo vedremo che esiste una connessione stretta tra regole (Rule) e funzioni.  
Prima di proseguire, però, dobbiamo studiare un nuovo tipo di regola.

#### ▫ 6.2.2. Le definizioni di funzione sono regole : i DownValues

#### ▫ 6.2.2. Le definizioni di funzione sono regole

Quando definiamo una funzione **f** in **Mathematica**, stiamo dunque definendo una Regola, che viene applicata globalmente (nel contesto `Global).

(Esempio. Rivediamo la definizione della funzione Fattoriale, qui in programmazione funzionale).

Quando il Kernel di **Mathematica** trova un match tra una espressione ed una Regola Globale, rimpiazza l'espressione con il RHS della Regola.

In effetti (in maniera semplificata) possiamo pensare al modo in cui il Kernel valuta una espressione come ad una unica applicazione dell'operatore ReplaceRepeated//. come segue:

```
(* Il Kernel valuta expression applicandole
{all global rules} con ReplaceRepeated *)
expression //.{all global rules}
```

In altre parole, le Regole Globali continuano ad essere applicate fino a che l'espressione non cambia più.

```
?ReplaceRepeated
expr = x^2 + y^6;
expr //.{x → 2 + a, a → 3} // TraditionalForm
```

Symbol i

*expr //.* rules repeatedly performs replacements until *expr* no longer changes.

ReplaceRepeated[*rules*] represents an operator

form of ReplaceRepeated that can be applied to an expression.

$y^5 + 25$

ReplaceRepeated equivale a ripetere ReplaceAll, fino ad arrivare ad una forma di punto fisso.

```
(* ReplaceRepeated equivale a ripetere ReplaceAll fino ad un punto fisso *)
?ReplaceAll
expr = x^2 + y^6;
passo1 = expr /. {x → 2 + a, a → 3} // TraditionalForm
(* al passo 1, la regola a→3 non viene applicata,
perche' in x^2+y^6 non c'e' il pattern "a" *)
passo2 = passo1 /. {x → 2 + a, a → 3} // TraditionalForm
passo3 = passo2 /. {x → 2 + a, a → 3};
passo3 == passo2
```

## Symbol



*expr* /. *rules* or ReplaceAll[*expr*, *rules*] applies a rule or list of rules in an attempt to transform each subpart of an expression *expr*.  
 ReplaceAll[*rules*] represents an operator form of ReplaceAll that can be applied to an expression.

$$(a + 2)^2 + y^6$$

$$y^6 + 25$$

True

Note su ReplaceRepeated.

Se viene dato un insieme di regole "circolare", allora ReplaceRepeated continuera' a dare risultati differenti (forever).

Nella pratica, pertanto, viene definito un numero massimo di iterazioni per ReplaceRepeated, determinato dall'Opzione MaxIterations.

Se vogliamo che ReplaceRepeated prosegua il piu' a lungo possibile, possiamo imporre MaxIterations → Infinity

(ma se il processo entra in loop, dovremo esplicitamente interrompere il run di Mathematica):

**ReplaceRepeated[*expr*, *rules*, MaxIterations → *Infinity*]**

⌘ Vediamo, con un altro esempio, la differenza tra ReplaceAll e ReplaceRepeated

```
log[a b c d] /. log[x_ y_] → log[x] + log[y]
log[a b c d] // . log[x_ y_] → log[x] + log[y]
```

```
log[a] + log[b c d]
```

```
log[a] + log[b] + log[c] + log[d]
```

Sia ReplaceAll che ReplaceRepeated applicano ogni regola a tutte le sottoparti dell'espressione originale

⌘ Se voglio lavorare su specifiche sottoparti di un'espressione, posso usare Replace e specificare il livello di applicazione.

Oppure, posso usare ReplacePart:

```
(* Se voglio lavorare su sottoparti,
posso specificare il livello di applicazione *)
Replace[1+x^2, x^2 → a+b, {1}]
```

```
1 + a + b
```

```
(* ReplacePart[ expr, i → new ] restituisce una espressione in
cui la parte i-esima di expr e' stata rimpiazzata da new *)
ReplacePart[{a, b, c, d, e}, 3 → xxx]
? ReplacePart
```

```
{a, b, xxx, d, e}
```

Symbol	<i>i</i>
ReplacePart[ <i>expr</i> , <i>i</i> → <i>new</i> ] yields an expression in which the $i^{\text{th}}$ part of <i>expr</i> is replaced by <i>new</i> .	
ReplacePart[ <i>expr</i> , { <i>i</i> <sub>0</sub> → <i>new</i> <sub>0</sub> , <i>i</i> <sub>1</sub> → <i>new</i> <sub>1</sub> , ...}] replaces parts at positions <i>i<sub>m</sub></i> by <i>new<sub>m</sub></i> .	
ReplacePart[ <i>expr</i> , { <i>i</i> , <i>j</i> , ...} → <i>new</i> ] replaces the part at position { <i>i</i> , <i>j</i> , ...}.	
ReplacePart[ <i>expr</i> , {{ <i>i</i> <sub>0</sub> , <i>j</i> <sub>0</sub> , ...} → <i>new</i> <sub>0</sub> , ...}] replaces parts at positions { <i>i<sub>m</sub></i> , <i>j<sub>m</sub></i> , ...} by <i>new<sub>m</sub></i> .	
ReplacePart[ <i>expr</i> , {{ <i>i</i> <sub>0</sub> , <i>j</i> <sub>0</sub> , ...}, ...} → <i>new</i> ] replaces all parts at positions { <i>i<sub>m</sub></i> , <i>j<sub>m</sub></i> , ...} by <i>new</i> .	
ReplacePart[ <i>i</i> → <i>new</i> ] represents an operator	
form of ReplacePart that can be applied to an expression.	

⌘ C'e' pure la ReplaceList, che applica la trasformazione sulla espressione originale intera, applicando una Regola oppure una LISTA di Regole (in tutti i modi possibili).

ReplaceList restituisce una lista di tutti i possibili risultati ottenuti.

```
ruleList = {x → a, x → b, x → c};
(* Replace usa solo la PRIMA regola applicabile *)
Replace[x, ruleList]
(* ReplaceList usa TUTTE le regole applicabili *)
ReplaceList[x, ruleList]
?ReplaceList
```

a

{a, b, c}

Symbol

i

ReplaceList[*expr*, *rules*] attempts to transform the entire expression *expr* by applying a rule or list of rules in all possible ways, and returns a list of the results obtained.

ReplaceList[*expr*, *rules*, *n*] gives a list of at most *n* results.

ReplaceList[*rules*] is an operator form of ReplaceList that can be applied to an expression.

▼

⌘ Vediamo un altro esempio di differenza tra Replace e ReplaceList

```
test = {a, b, c, d, e, f};
(* questa regola dice di sostituire ad ogni lista
{x_, y_} una lista { {x}, {y} } di due sottoliste *)
regola = {x_, y_} → {{x}, {y}};
(* Replace usa solo la PRIMA sostituzione applicabile *)
Replace[test, regola]
(* ReplaceList usa TUTTE le sostituzioni applicabili *)
ReplaceList[test, regola]
```

{{a}, {b, c, d, e, f}}

{{{a}, {b, c, d, e, f}}, {{a, b}, {c, d, e, f}},
{{a, b, c}, {d, e, f}}, {{a, b, c, d}, {e, f}}, {{a, b, c, d, e}, {f}}}

Secondo Roman Maeder (autore di vari testi su “Programming in Mathematica”), il meccanismo di patter-matching, abbinato al meccanismo di sostituzione di regole, e' alla base di qualsiasi altro stile di programmazione.

Questo ed il fatto che ogni cosa (in Mathematica) e' rappresentata come atomo o come espressione hanno per conseguenza la flessibilita' di programmazione in Mathematica.

# Patterns

Patterns are a fundamental concept in *Mathematica*.

The pattern `_` is called "blank" and `_` stands for "anything".

## MatchQ

**MatchQ** tests whether something matches a **pattern**.

```
(* Any list of 3 elements, with x in the middle,
matches the pattern {_, x, _} *)
Clear[a, b, c, d, f, x];
pattern3 = {_, x, _};
l1 = {a, x, b}; mq1 = MatchQ[l1, pattern3];
l2 = { {a, c}, x, {b, d, f} }; mq2 = MatchQ[l2, pattern3];
l3 = { {a, b, c}, x, b }; mq3 = MatchQ[l3, pattern3];
{mq1, mq2, mq3}
{True, True, True}

(* No match, because x is not the 2nd element *)
Clear[a, b, c, z, x];
pattern3 = {_, x, _};
l4 = {a, z, b}; mq4 = MatchQ[l4, pattern3];
l5 = {a, b, x}; mq5 = MatchQ[l5, pattern3];
l6 = {{a, x, c}, a, b}; mq6 = MatchQ[l6, pattern3];
{mq4, mq5, mq6}
{False, False, False}

(* No match, because there are more than 3 elements *)
Clear[a, b, c, d, x];
pattern3 = {_, x, _};
l7 = {a, b, x, c, d}; mq7 = MatchQ[l7, pattern3];
mq7
False
```

```
(* Any list with 2 elements matches the pattern {_,_} *)
Clear[a, b, c, d, f];
pattern2 = {_, _};
l1 = {a, a}; mq1 = MatchQ[l1, pattern2];
l2 = { {a, b, c}, {d, f} }; mq2 = MatchQ[l2, pattern2];
(* No match, because there are more than 2 elements *)
l3 = {a, a, a}; mq3 = MatchQ[l3, pattern2];
{mq1, mq2, mq3}
{True, True, False}
```

## Cases (1)

**MatchQ** lets you test one thing at a time against a pattern.

**Cases** lets you pick out all the elements (“cases”) in a list that match a pattern.

```
Clear[a, b, c];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a}, {b, b, b}, {b}};

(* Find all elements that match the pattern {_,_} *)
pattern0 = {_, _};
Cases[ testList, pattern0 ]
{{a, a}, {b, a}, {b, b}, {c, a}}

(* Blank:      Find all elements that match { b , 1 element } *)
pattern1 = {b , _};
Cases[ testList, pattern1 ]
{{b, a}, {b, b}}

(* Double Blank:    Find all elements that match { b , some elements } *)
pattern2 = {b , __};
Cases[ testList, pattern2 ]
{{b, a}, {b, b}, {b, b, b}}

(* Triple Blank:    Find all elements that match { b , optional } *)
pattern3 = {b , ___};
Cases[ testList, pattern3 ]
{{b, a}, {b, b}, {b, b, b}, {b}}
```

MatcQ can be used to test whether each element matches { b , \_ }

```

Clear[a, b, c];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a}, {b, b, b}, {b}};
pattern1 = {b, _};

map1 = Map[
  MatchQ[#, pattern1] &,
  testList]
{False, True, False, True, False, False}

```

## Select

**Select** what **matches** gives the same result as **Cases**

```

(* Here, we use same testList as before, same pattern1,
and same pure function used in map1: MatchQ[ # ,pattern1 ] & *)
Clear[a, b, c];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a}, {b, b, b}, {b}};
pattern1 = {b, _};

sel = Select[
  testList,
  MatchQ[#, pattern1] &
]

sel == Cases[ testList, pattern1 ]
{{b, a}, {b, b}}
True

```

## Cases (2)

In a pattern,  $a|b$  indicates "either a **Or** b".

It is called **Alternatives**[a,b] (see: guide/Patterns).

It is a pattern object that represents any of the patterns a, b

```

Clear[a, b, c];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a}, {b, b, b}, {b}};
pattern = {a | b, _};
Cases[ testList, pattern ]
{{a, a}, {b, a}, {b, b}}

```

Note (see Q&A below).

**Alternatives**[a,b] is different from **Or**[a,b] i.e.  $a||b$

**Or**[a,b, ...] is used to define/combine assertions, assumptions, conditions,

systems of equations/inequalities, sets given by algebraic conditions  
(see: guide/LogicAndBooleanAlgebra)

```
(* Or[a,b] is a logical operator .
It evaluates its arguments in order,
giving True immediately if any of them are True or False,
and False if they are all False. *)
```

## Example

Create a **list**, then pick out elements that match particular **patterns**.

```
(* List: 3-digit numbers from 100 to 500, with step 55 *)
input = Range[100, 500, 55]
digits = IntegerDigits[input]
{100, 155, 210, 265, 320, 375, 430, 485}

{{1, 0, 0}, {1, 5, 5}, {2, 1, 0}, {2, 6, 5}, {3, 2, 0}, {3, 7, 5}, {4, 3, 0}, {4, 8, 5}}

(* Pattern: 3-digit numbers whose last digit is 5 *)
idPattern5 = {_, _, 5};
output5 = Cases[digits, idPattern5]
Map[FromDigits, output5]
{{1, 5, 5}, {2, 6, 5}, {3, 7, 5}, {4, 8, 5}}

{155, 265, 375, 485}

(* Pattern: 3-digit numbers whose second digit is 1 or 2 *)
idPattern12 = {_, 1 | 2, _};
output12 = Cases[digits, idPattern12]
Map[FromDigits, output12]
{{2, 1, 0}, {3, 2, 0}},

{210, 320}
```

## Single, Double, Triple Blank

```

Clear[a, b, c, d];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a},
    {a, c, b}, {b, b, b}, {a}, {b}, {c}, {d}, {}};
(* find sequences ending with b .... *)
(* .... and beginning with 1 element *)
pattern1 = {_, b};
Cases[ testList, pattern1 ]
(* .... and beginning with 1 or more elements *)
pattern2 = {_, b};
Cases[ testList, pattern2 ]
(* .... and beginning with none, 1 or more elements *)
pattern3 = {___, b};
Cases[ testList, pattern3 ]
{{b, b}}
{{b, b}, {a, c, b}, {b, b, b}}
{{b, b}, {a, c, b}, {b, b, b}, {b}}


Clear[a, b, c, d];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a},
    {b, b, b}, {a}, {b}, {c}, {d}, {}};
(* find sequences ending with a or b , or beginning with c ... *)
(* .... and with another element *)
patternC1 = {_, a | b } | {c, ___};
Cases[ testList, patternC1 ]
(* .... and with 1 or more other elements *)
patternC2 = {___, a | b } | {c, ___};
Cases[ testList, patternC2 ]
(* .... and with other optional elements *)
patternC3 = {___, a | b } | {c, ___};
Cases[ testList, patternC3 ]
{{a, a}, {b, a}, {b, b}, {c, a}}
{{a, a}, {b, a}, {b, b}, {c, a}, {b, b, b}}
{{a, a}, {b, a}, {b, b}, {c, a}, {b, b, b}, {a}, {b}, {c}}

```

## Pattern Head[\_]

Patterns are not just about lists; they can involve anything.

```

Clear[a, b, c, f, g, x, y];
test = {f[1], f[2, 3], g[1], g[2, 3],
{a, b, c}, {a, f, c}, {}, {a, f[y], c},
f[x], f[x, x], f[], f[a],
g[x], g[x, x], g[], g[a]};
(* Pick out cases that match the pattern f[_] *)
patternF1 = f[_];
Cases[test, patternF1]
(* Pick out cases that match the pattern f[_] *)
patternF2 = f[_];
Cases[test, patternF2]
(* Pick out cases that match the pattern f[___] *)
patternF3 = f[___];
Cases[test, patternF3]
{f[1], f[x], f[a]}
{f[1], f[2, 3], f[x], f[x, x], f[a]}
{f[1], f[2, 3], f[x], f[x, x], f[], f[a]}

Clear[a, b, c, f, g, h, x, y];
test = {f[1], f[2, 3], g[1], g[2, 3],
{a, b, c}, {a, f, c}, {}, {h}, h, {a, f[y], c},
f[x], f[x, x], f[], f[a],
g[x], g[x, x], g[], g[a]};
(* Pick out cases that match the pattern _[_] *)
patternH1 = _[_];
Cases[test, patternH1]
(* Pick out cases that match the pattern _[___] *)
patternH2 = _[___];
c2 = Cases[test, patternH2]
(* Pick out cases that match the pattern _[___] *)
patternH3 = _[___];
c3 = Cases[test, patternH3];
Sort[test] == Sort[Append[c3, h]]
(* patternH3 matches everything in test, but atom h *)
{f[1], g[1], {h}, f[x], f[a], g[x], g[a]}
{f[1], f[2, 3], g[1], g[2, 3], {a, b, c}, {a, f, c},
{h}, {a, f[y], c}, f[x], f[x, x], f[a], g[x], g[x, x], g[a]}
True

```

## ReplaceAll (./)

One of the many uses of patterns is to define replacements

```

Clear[a, b, mylist, myrule];
mylist = {a, b, a, a, b, b, a, b} ;
(* myrule = Rule[b, Red] *)
myrule = b → Red;
(* mylist /. myrule *)
ReplaceAll[ mylist, myrule ]
{a, █, a, a, █, █, a, █}

Clear[a, b, c, mylist, myrule];
mylist = {{1, a}, {1, b}, {1, a, b}, {2, b, c}, {2, b}} ;
myrule = {1, _} → Red;
ReplaceAll[ mylist, myrule ]
{█, █, {1, a, b}, {2, b, c}, {2, b}}

Clear[a, b, c, mylist, myrules];
mylist = {{1, a}, {1, b}, {1, c}, {1, a, b}, {2, b, c}, {2, b}, {1}, {b}, {c}} ;
myrules = {
    {1, _} → Red,
    {_, b} → Yellow ,
    {_, c} → Orange
    } ;
ReplaceAll[ mylist, myrules ]
{█, █, █, █, █, █, {1}, {b}, █}

```

```

Clear[a, b, c, mylist, myrules, myrules2];
mylist = {{1, a}, {1, b}, {1, c}, {1, a, b}, {2, b, c}, {2, b}, {1}, {b}, {c}} ;
myrules = {
  {1, _} → Red,
  {_, b} → Yellow ,
  {__, c} → Orange
} ;
myrules2 = RotateRight[myrules] ;
TableForm[myrules2]
(* myrules3=RotateLeft[myrules] *)
ReplaceAll[ mylist , myrules2]
(* ReplaceAll[ mylist , myrules3 ] *)
{__, c} → █
{1, _} → █
{_, b} → █
{█, █, █, █, █, █, {1}, {b}, █}

```

## Named\_pattern

The "blank" pattern `_` matches anything.

For example, `{_, _}` matches any list of two elements.

But what if you want to insist that the two elements be the same?

You can do that using a pattern like `{x_, x_}`.

```

Clear[a, b, c, mytest, mypatternAny, mypattern, mypatternX];
mytest = {{a, a, a}, {a, a}, {a, b}, {a, c}, {b, a}, {b, b}, {a}, {b}, {c}, {x, x}};
mypatternAny = {_, _};
mypattern = {x_, x_};
mypatternX = {x, x};
Cases[mytest, mypatternAny]
Cases[mytest, mypattern]
Cases[mytest, mypatternX]
{{a, a}, {a, b}, {a, c}, {b, a}, {b, b}, {x, x}}
{{a, a}, {b, b}, {x, x}}
{{x, x}}

```

`x_` is an example of a **named** pattern.

Named patterns are important in replacements,  
because they give a way for making use of Parts of what one is replacing.

```

Clear[a, mycolors, mypatternYellow];
a = 3;
(* swatches *)
mycolors = { {1, Red},
             {1, Blue},
             {1, Cyan, Pink},
             {Purple, 1, Brown},
             {Green, Magenta, 1},
             {2, Orange} ,
             {1, a},
             {1, 4},
             {1, b},
             {1, b} /. b → 5,
             HoldForm[{1, b} /. b → 5] }

(* Replace 2-component sublists of the form {1,x} with {x,x,Yellow,x,x} *)
mypatternYellow = {1, x_} → {x, x, Yellow, x, x};
output = ReplaceAll[ mycolors, mypatternYellow ]

{{1, \textcolor{red}{\boxed{\textcolor{red}{1}}}}, {1, \textcolor{blue}{\boxed{\textcolor{blue}{1}}}}, {1, \textcolor{cyan}{\boxed{\textcolor{cyan}{1}}}, \textcolor{pink}{\boxed{\textcolor{pink}{1}}}}, {\textcolor{purple}{\boxed{\textcolor{purple}{1}}}, 1, \textcolor{brown}{\boxed{\textcolor{brown}{1}}}},
 {\textcolor{green}{\boxed{\textcolor{green}{1}}}, \textcolor{magenta}{\boxed{\textcolor{magenta}{1}}}, 1}, {2, \textcolor{orange}{\boxed{\textcolor{orange}{1}}}}, {1, 3}, {1, 4}, {1, b}, {1, 5}, {1, b} /. b → 5}

{\textcolor{red}{\boxed{\textcolor{red}{1}}}, \textcolor{red}{\boxed{\textcolor{red}{1}}}, \textcolor{yellow}{\boxed{\textcolor{yellow}{1}}}, \textcolor{red}{\boxed{\textcolor{red}{1}}}, {\textcolor{blue}{\boxed{\textcolor{blue}{1}}}, \textcolor{blue}{\boxed{\textcolor{blue}{1}}}, \textcolor{yellow}{\boxed{\textcolor{yellow}{1}}}, \textcolor{blue}{\boxed{\textcolor{blue}{1}}}, \textcolor{blue}{\boxed{\textcolor{blue}{1}}}}, {1, \textcolor{cyan}{\boxed{\textcolor{cyan}{1}}}, \textcolor{pink}{\boxed{\textcolor{pink}{1}}}}, {\textcolor{purple}{\boxed{\textcolor{purple}{1}}}, 1, \textcolor{brown}{\boxed{\textcolor{brown}{1}}}}, {\textcolor{green}{\boxed{\textcolor{green}{1}}}, \textcolor{magenta}{\boxed{\textcolor{magenta}{1}}}, 1}, {2, \textcolor{orange}{\boxed{\textcolor{orange}{1}}}},
 {3, 3, \textcolor{yellow}{\boxed{\textcolor{yellow}{1}}}, 3, 3}, {4, 4, \textcolor{yellow}{\boxed{\textcolor{yellow}{1}}}, 4, 4}, {b, b, \textcolor{yellow}{\boxed{\textcolor{yellow}{1}}}, b, b}, {5, 5, \textcolor{yellow}{\boxed{\textcolor{yellow}{1}}}, 5, 5}, {b, b, \textcolor{yellow}{\boxed{\textcolor{yellow}{1}}}, b, b} /. b → 5}

```

## Rule

The form  $x \rightarrow b$  is **Rule[x, b]** (\* RuleDelayed  $\Rightarrow$  ossia  $:>$  \*)

If  $x_$  appears on the left-hand side of a Rule ,

then whatever  $x_$  matches can be referred to (on the right - hand side) as x.

```

Clear[a, f, g];
functions = { f[1], g[2], f[2], g[3], f[], f[2, 3] };
myrule = f[x_] → x + 10 ;
(* functions /. myrule *)
ReplaceAll[ functions , myrule]
(* f[1] → 1 + 10 *)
(* g[2] → no match with f[ x_ ] *)
(* f[2] → 2 + 10 *)
(* g[3] → no match with f[ x_ ] *)
(* f[ ] → no match with f[ x_ ] *)(* f[2,3] → no match with f[ x_ ] *)
{11, g[2], 12, g[3], f[], f[2, 3]}

```

You can use Rules inside **Cases** as well.

Below, Cases picks out elements in the list that match  $f[x_]$  ,  
and gives the result of replacing them by  $x+10$ :

```

Clear[f, g];
functions = { f[1], g[2], f[2], g[3], f[], f[2, 3] } ;
myrule = f[x_] → x + 10 ;
Cases[ functions , myrule ]
{11, 12}

```

---

## Vocabulary

_	pattern standing for anything (“blank”)
__	pattern standing for any sequence (“double blank”)
___	pattern standing for any optional sequence (“triple blank”)
x_	pattern named x
a b	pattern matching a or b
<b>MatchQ</b> [expr,pattern]	test whether expr matches a pattern
<b>Cases</b> [list,pattern]	find cases of a pattern in a list
<b>lhs</b> → <b>rhs</b>	<b>Rule</b> for transforming lhs into rhs
<b>expr</b> . <b>lhs</b> → <b>rhs</b>	<b>ReplaceAll</b> lhs by rhs in expr
<b>HoldForm</b>	
<b>MaxIterations</b>	
<b>Nothing</b>	

---

## Exercises

**32.1** Find numbers with 3 **or more** digits, beginning with 1, ending with 9, in Range[10^3]+10

**32.2** Find lists of numbers made of exactly 3 **identical** digits in Range[10^4].

**32.3** In the digit lists for the first 1000 squares, find those that begin with 9 and end with 0 or 1.

**32.4** In IntegerDigits[Range[30]] replace all 0's by Gray and all 9's by Orange.

**32.5** Make a list of the digits of 2^100, replacing all zeros by Red.

**32.6** Remove vowels (a, e, i, o, u) from the list of Characters in “Computational Mathematics” (Nothing).

**32.7** Use Cases to obtain same output as: Select[ IntegerDigits[2^100], #==

0||#==1 & ] .

**32.8** Use Cases to obtain same output as: `Select[ IntegerDigits[Range[800, 999]], First[#]==Last[#] & ]` .

---

## Q & A

---

## Tech Notes

---

## More to Explore

Guide to Patterns in *Mathematica*.

`Hyperlink["https://reference.wolfram.com/language/guide/Patterns.html"]`

<https://reference.wolfram.com/language/guide/Patterns.html>

La prof testerà il progetto con cose che non hanno senso per rompere il sito e vedere le falte —> se rompe il codice, non deve ricevere messaggi di errore (cosa che succederà se non abbiamo previsto certi casi)

### Esercizio 2 di 6.2.3

Voglio usare solo Part per simulare Take.

Definisco myTake per casi (definizioni multiple).

Nel Caso Generale, in particolare, voglio avere 3 argomenti di chiamata:

il primo argomento (m, start) obbligatorio oppure opzionale;

il secondo argomento (n, stop) obbligatorio oppure opzionale;

il terzo argomento (s, step) opzionale.

```
(* genero una lista di interi da -5 a 5 *)
```

```
listatest = Range[-5, 5];
```

```
? Take
```

Symbol i

Take[list, n] gives the first  $n$  elements of *list*.  
Take[list, -n] gives the last  $n$  elements of *list*.  
Take[list, {m, n}] gives elements  $m$  through  $n$  of *list*.  
Take[list, seq<sub>1</sub>, seq<sub>2</sub>, ...] gives a nested list  
in which elements specified by seq<sub>i</sub> are taken at level  $i$  in *list*.

Documentation [Local »](#) | [Web »](#)  
Attributes {NHoldRest, Protected}  
Full Name System`Take

```
? Part
```

Symbol i

expr[[i]] or Part[expr, i] gives the  $i^{\text{th}}$  part of *expr*.  
expr[[-i]] counts from the end.  
expr[[i, j, ...]] or Part[expr, i, j, ...] is equivalent to expr[[i]][[j]] ....  
expr[[{i<sub>1</sub>, i<sub>2</sub>, ...}]] gives a list of the parts  $i_1, i_2, \dots$  of *expr*.  
expr[[m ; n]] gives parts  $m$  through  $n$ .  
expr[[m ; n ; s]] gives parts  $m$  through  $n$  in steps of  $s$ .  
expr[["key"]] gives the value associated with the key "key" in an association *expr*.  
expr[[Key[k]]] gives the value associated with an arbitrary key  $k$  in the association *expr*.

(VERSIONE 1a con argomenti opzionali posizionali)

### VERSIONE 1 con argomenti opzionali posizionali (: ossia Optional[ ])

```

Clear[myTake]
(* Caso: Singoletto *)
myTake[x_List, {n_}] := {x[[n]]};

(* Se n ≥ 0, prendo i primi n elementi, da 1 ad n, della lista x (con passo 1) *)
(* myTake[x_List,n_]/;n≥0:=x[[1;;n;;1]]; *)
(* myTake[x_List,n_]/;n≥0:=x[[;;n]]; *)
myTake[x_List, n_] /; n ≥ 0 := x[[1 ;; n]];

(* Se n < 0, prendo gli ultimi n elementi dalla lista x *)
(* myTake[x_List,n_]/;n<0:=x[[Length[x]+n+1;;Length[x];;1]]; *)
(* myTake[x_List,n_]/;n<0:=x[[Length[x]+n+1;;]]; *)
myTake[x_List, n_] /; n < 0 := x[[Length[x]+n+1 ;; Length[x]]];

(* Caso Generale, con passo s opzionale ed {m,n,s} tra graffe .
   Intercetta le chiamate myTake[x, {m,n}] e myTake[x,{m,n,s}]*)
myTake[x_List, {m_Integer, n_Integer, s_Integer : 1}] := x[[m ;; n ;; s]];
(* Possiamo arricchirlo, rendendo opzionali due o tutti gli argomenti *)
(* myTake[x_List,m_Integer:1, n_Integer, s_Integer:1]:=x[[m;;n;;s]]; *)
(* myTake[x_List,m_Integer:1, n_Integer:All, s_Integer:1]:=x[[m;;n;;s]]; *)
(* La chiamata myTake[x, {n}] viene intercettata dal Caso Singoletto *)

(* Caso con tre argomenti opzionali, senza graffe.
   Intercetta la chiamata myTake[x] *)
(* myTake[x_List,m_Integer:1,n_Integer:-1,s_Integer:1]:=x[[m;;n;;s]]; *)
myTake[x_List, m_Integer : 1, n_Integer : All, s_Integer : 1] := x[[m ;; n ;; s]];
modo posizionale -> diamo un nome ai vari argomenti ma è importante la loro posizione
{myTake[listatest, 2] == Take[listatest, 2],
 myTake[listatest, -3] == Take[listatest, -3],
 myTake[listatest, {2}] == Take[listatest, {2}],
 myTake[listatest, {-3}] == Take[listatest, {-3}],
 myTake[listatest] == Take[listatest],
 myTake[listatest, {2, 4}] == Take[listatest, {2, 4}],
 myTake[listatest, {-4, -2}] == Take[listatest, {-4, -2}],
 myTake[listatest, {2, -4}] == Take[listatest, {2, -4}],
 myTake[listatest, {3, 7, 2}] == Take[listatest, {3, 7, 2}]}

{True, True, True, True, True, True, True, True}

```

## VERSIONE 2 con argomenti opzionali nominali (named Options)

```

Clear[gsTake]
(* Caso: Singoletto *)
gsTake[x_List, {n_}] := {x[[n]]};

(* Se n ≥ 0, prendo i primi n elementi, da 1 ad n, della lista x (con passo 1) *)
(* myTake[x_List,n_]/;n≥0:=x[[1;;n;;1]]; *)
gsTake[x_List, n_]/; n ≥ 0 := x[[1 ;; n]];

(* Se n < 0, prendo gli ultimi n elementi dalla lista x *)
(* myTake[x_List,n_]/;n<0:=x[[Length[x]+n+1;;Length[x];;1]];*)
gsTake[x_List, n_]/; n < 0 := x[[Length[x]+n+1 ;; Length[x]]];

(* Caso Generale con argomenti opzionali *)
(* Options[gsTake]={m→1,n→-1,s→1}; *)
Options[gsTake] = {m → 1, n → All, s → 1};
gsTake[x_List, OptionsPattern[]] :=
  x[[ OptionValue[m] ;; OptionValue[n] ;; OptionValue[s] ]];

{gsTake[listatest, 2] == Take[listatest, 2],
 gsTake[listatest, -3] == Take[listatest, -3],
 gsTake[listatest, {2}] == Take[listatest, {2}],
 gsTake[listatest, {-3}] == Take[listatest, {-3}],
 gsTake[listatest] == Take[listatest],
 (* Sintassi diversa tra gsTake e Take *)
 gsTake[listatest, m → 2, n → 4] == Take[listatest, {2, 4}],
 gsTake[listatest, m → -4, n → -2] == Take[listatest, {-4, -2}],
 gsTake[listatest, m → 2, n → -4] == Take[listatest, {2, -4}],
 gsTake[listatest, m → 3, n → 7, s → 2] == Take[listatest, {3, 7, 2}]}

{True, True, True, True, True, True, True, True}

```

Riferimento : tutorial Patterns, sezioni:

- Setting Up Functions with Optional Arguments
- Optional and Default Arguments

<https://reference.wolfram.com/language/tutorial/Patterns.html#17673>

## Programmazione basata su Regole

### ■ 6.3. Mattoni per costruire Pattern \*

Mathematica fornisce un ricco insieme di "mattoni" per la costruzione di pattern.

In questo testo non e' possibile illustrarli tutti: verranno presentati solo alcuni esempi.

Per una piu' ampia documentazione, si rimanda a (capitoli successivi di questo testo, come pure a) molti altri testi della letteratura collegata all'ambiente di Mathematica.

#### □ 6.3.1 Pattern di vincolo

Tre costrutti possono essere usati come **pattern di vincolo**.

Due di essi vincolano i valori del **Blank singolo**.

Il terzo puo' essere usato per specificare la **relazione** tra differenti variabili—pattern.

##### head

Un Blank puo' essere vincolato a combaciare solo con certe espressioni aventi un Head particolare (cfr. § 4.1.3 "Fare un controllo di tipo") appendendo un Head al Blank stesso.

```
(* Il pattern _Integer combacia solo con interi *)
Cases[{1, Sqrt[2], b}, _Integer]
{1}
```

##### ?test

Un Blank puo' essere vincolato dalla forma \_?test, in cui "test" puo' essere qualsiasi funzione di un solo argomento.

Se l'applicazione del test (alla espressione combaciante col Blank) restituisce il valore True, allora il pattern combacia.

```
(* Secondo modo di scrivere un pattern che combacia solo con interi *)
Cases[{1, Sqrt[2], b}, _?IntegerQ]
{1}
```

Potendo scegliere, e' piu' efficiente far combaciare la Head strutturalmente usando \_Integer piuttosto che usare \_?IntegerQ,

per lo meno in un semplice esempio come quello appena visto.

D'altra parte, esistono molti predicati (cfr. § 3.4) che permettono di testare condizioni complesse, quali, ad esempio, il fatto che un'espressione sia o meno un Atomo (AtomQ) oppure un numero (NumberQ), per i quali e' utile la seconda sintassi.

Oltre ad usare predicati definiti da sistema, ovviamente, e' possibile definirne di propri.

Vediamo un esempio.

Costruiamo un predicato "between", che useremo nella forma `_?between` per testare se un'espressione e' un numero tra 1 e 10.

```
(* Costruisco una funzione da usare nella Cases in forma di predicato *)
between[x_] := 1 ≤ x ≤ 10
(* costruisco una tabella di 20 numeri random tra 1 e 100 *)
SeedRandom[8];
tabella = Table[ RandomReal[{1, 100}], {20}];
Sort[tabella]
(* Qui il pattern _?between combacia con un'
espressione solo se essa e' un numero tra 1 e 10.
Quindi Cases cerca, nella tabella, eventuali numeri tra 1 e 10 *)
Cases[ tabella , _?between]

{1.88952, 6.8186, 14.7483, 16.476, 18.684, 18.6946,
 20.7307, 21.4937, 22.1198, 29.5493, 29.5974, 45.4095, 48.6137,
 49.0859, 49.9518, 75.7534, 80.5866, 89.1365, 89.7225, 91.3095}
```

```
{1.88952, 6.8186}
```

E' possibile costruire un pattern che **combini assieme** il far combaciare la Head (e.g. `_Integer`) con una funzione di test (e.g. `_?NonNegative`).

Nell'esempio che segue, il pattern combacia solo con interi non negativi.

```
test = {-3, -1, 0, 1, 1.5, 2, 2.5};
Cases[ test , _Integer?NonNegative]

{0, 1, 2}
```

Nell'esempio che segue, il pattern combacia solo con reali non negativi.

```

SeedRandom[8];
tabella2 = Table[ RandomReal[{-50, 50}], {20} ];
Sort[tabella2]
Cases[ tabella2 , _Real?NonNegative]

{-49.1015, -44.1226, -36.1128, -34.3677, -32.1374, -32.1267,
 -30.07, -29.2993, -28.6669, -21.1623, -21.1137, -5.14196, -1.90533,
 -1.42839, -0.553752, 25.5085, 30.3905, 39.0268, 39.6187, 41.2217}

```

```
{39.6187, 39.0268, 30.3905, 41.2217, 25.5085}
```

**pattern /; condition** —> usata per evitare di usare if/else (ma utilizza gli if/else)

Una condizione consiste in una chiamata alla Condition ( /; ) seguita da una qualsiasi espressione che coinvolga variabili—pattern.

Essa puo' essere attaccata, praticamente, a qualsiasi parte di un pattern.

Ad esempio, una qualsiasi delle regole che seguono puo' essere usata per definire (in modo ricorsivo) la funzione fattoriale (per numeri interi e non negativi):

```

body[n_] := Apply[ Times , Range[n]];    la forma più carina è la 1 (la prof eviterebbe la 5 perchè è ridondante)

factorial1[n_Integer?NonNegative] := body[n]

factorial2[n_Integer /; NonNegative[n]] := body[n]

factorial3[n_Integer] /; NonNegative[n] := body[n]

factorial4[n_Integer] := body[n] /; NonNegative[n]

factorial5[n_] := body[n] /; IntegerQ[n] && NonNegative[n]

{factorial1[n], factorial2[n], factorial3[n], factorial4[n], factorial5[n]} /. n → 5

{120, 120, 120, 120, 120}

```

Le Condition sono utili perche' permettono di ovviare alla necessita' di scrivere una funzione test separata.

Ad esempio, la funzionalita' della "between" (che testa una lista di numeri per considerare solo quelli compresi tra 1 e 10)

puo' essere incorporata (come Condition) nella definizione di una funzione, come segue.

```
(* Cosi' definita, la funzione f accetta solo argomenti tra 1 e 10 ,
quindi restituisce il fattoriale solo per tali argomenti *)
Clear[body, f, x, n, tabella];

body[n_] := Apply[Times, Range[n]];

f[x_ /; 1 <= x <= 10] := body[x];

SeedRandom[2];
tabella = Module[{ntemp, ftemp},
  Table[
    ntemp = RandomInteger[{-10, 10}];
    ftemp = f[ntemp];
    {ntemp, ftemp},
    {5}]
  ];
TableForm[tabella, TableAlignments -> Center]
```

```
-7      f[-7]
8      40 320
1      1
0      f[0]
7      5040
```

```
(* Nota. Ricordo che il fattoriale di 0 vale 1 *)
Factorial[0]
```

```
1
```

Le Condition sono piu' flessibili dei costrutti **?test**,  
perche' esse possono coinvolgere piu' di una sola variabile—pattern alla volta,  
come mostrato nell'esempio che segue.

```

Clear[body, f, x, y, n];
body[x_] := Apply[Times, Range[x]];

(* La funzione f verra' chiamata solo se il suo
secondo argomento e' maggiore del primo argomento *)
f[x_, y_] /; x < y := body[x];

SeedRandom[2];
tabella = Module[{rtemp, stemp, ftemp},
  Table[
    rtemp = RandomInteger[{1, 10}];
    stemp = RandomInteger[{1, 20}];
    ftemp = f[rtemp, stemp];
    {rtemp, stemp, ftemp},
    {5}]
  ];
TableForm[tabella, TableAlignments → Center]

```

```

9   19    362 880
6   4     f[6, 4]
1   14    1
2   2     f[2, 2]
4   10   24

```

### Nota.

Nel caso particolare dell'esempio qui sopra, non e' possibile piazzare Condition cosi':

```
f[x_, y_] /; x < y := body[x]; (* NO! *)
```

Così facendo, il Kernel assume che noi si voglia testare la variabile—pattern  $y_$ —paragonandola ad un simbolo globale  $x$  (notiamo i colori verde e blu dei vari simboli), anziché paragonare tra loro gli argomenti  $x$  ed  $y$ .

```

Clear[body, f, x, y, n];
body[x_] := Apply[Times, Range[x]];

(* La funzione f verra' chiamata solo se
   il suo secondo argomento e' maggiore non del primo,
   bensic' di una variabile x globale, che pero' non e' settata *)
f[x_, y_ /; x < y] := body[x];

SeedRandom[2];
tabella = Module[{rtemp, stemp, ftemp},
  Table[
    rtemp = RandomInteger[{1, 10}];
    stemp = RandomInteger[{1, 20}];
    ftemp = f[rtemp, stemp];
    {rtemp, stemp, ftemp},
    {5}]
  ];
tabella // TableForm

9      19      f[9, 19]
6      4       f[6, 4]
1      14      f[1, 14]
2      2       f[2, 2]
4      10      f[4, 10]

```

⌘ Unico **vincolo su una condizione**: se la condizione coinvolge piu' di una sola variabile—pattern alla volta,  
allora la condizione stessa deve apparire fuori dalla piu' piccola espressione contenente tutte tali variabili—  
pattern.

```

(* SI! *) f[x_, y_] /; x < y := body[x];
(* NO! *) f[x_, y_ /; x < y] := body[x];

```

Per ragioni sia di efficienza, sia di leggibilita', e' meglio piazzare una condizione il piu' vicino possibile  
alla/alle variabile/i che la condizione stessa coinvolge.

#### □ Esercizio 1 pagina 154

Supponiamo che, in una funzione  $f[x,y]$ , noi si voglia paragonare il secondo argomento  $y$  contro una  $x$  globale

(e non contro il primo argomento di f).

Vediamo che cosa accade.

TASK. Ridefinire la funzione  $f[x,y]$  (che utilizza la Condition  $x < y$ ) come segue :

```
Clear[f, x, y, a, b, test0, test, tabella0, tabella];
(* Se il secondo argomento y e' maggiore non del primo argomento,
bensì' di una variabile Globale x, allora restituisci True,
altrimenti False *)
f[x_, y_] /; x < y := True
f[_, _] := False
```

Valutare poi f in vari argomenti.

In particolare, assegnare alla variabile x (intesa come globale) il valore 2 e valutare f[99,3].

⌘ Inizio valutando Clear[x], quindi x (come variabile Globale) non è assegnata .

```
(* Dato che alla variabile Globale x NON e' assegnato alcun valore,
il test x < y tra il secondo argomento y di f e tale variabile Globale NON
sara' mai verificato, quindi la chiamata ad f restituira' False *)
f[99, 3]
```

False

```
(* idem *)
test0 = {f[a, b], f[a], f[], f, f[0, 1], f[1, 0], f[0, 0], f[1, 1], f[2, 2], f[3, 3]}
{False, f[a], f[], f, False, False, False, False, False}
```

```
(* idem *)
SeedRandom[3];
tabella0 = Module[{rtemp, stemp, ftemp},
  Table[
    rtemp = RandomReal[{2, 50}];
    stemp = RandomReal[{2, 50}];
    ftemp = f[rtemp, stemp];
    {rtemp, stemp, ftemp},
    {5}]
];
tabella0 // TableForm
```

24.9706	2.41745	False
18.6574	8.68543	False
10.6689	27.3777	False
29.7722	38.4968	False
21.4063	45.3789	False

Ora assegno un valore alla variabile Globale x.

```
(* Ora assegno un valore alla variabile Globale x *)
x = 2;
(* Il secondo argomento di f e' y=3 ed e' maggiore della variabile Globale x=2,
quindi l'output sara' True,
anche se il secondo argomento y=
3 NON e' maggiore del primo argomento x_local=99 di f *)
f[99, 3]
```

True

```
test = {f[a, b], f[a], f[], f, f[0, 1], f[1, 0], f[0, 0], f[1, 1], f[2, 2], f[3, 3]}
(* Paragono test e test0 *)
test0;
test0 == test;
```

{False, f[a], f[], f, False, False, False, False, True}

```

SeedRandom[3];
tabella = Module[{rtemp, stemp, ftemp},
  Table[
    rtemp = RandomReal[{2, 50}];
    stemp = RandomReal[{2, 50}];
    ftemp = f[rtemp, stemp];
    {rtemp, stemp, ftemp},
    {5}]
  ];
tabella // TableForm
(* Paragono tabella e tabella0 *)
tabella0 // TableForm;
tabella0 == tabella;

24.9706 2.41745 True
18.6574 8.68543 True
10.6689 27.3777 True
29.7722 38.4968 True
21.4063 45.3789 True

```

TASK. Ridefinire la funzione  $f[x,y]$  (correttamente) in modo che il test avvenga tra i suoi due argomenti  $x, y$

⌘ Inizio valutando  $\text{Clear}[x]$ , quindi  $x$  (come variabile Globale) non è assegnata.

```

Clear[f, x, y, a, b, test0, test, tabella0, tabella];
(* Se il secondo argomento y e' maggiore del primo argomento,
allora restituisci True,
altrimenti False *)
f[x_, y_] /; x < y := True
f[_, _] := False

```

```
f[99, 3]
```

```
False
```

```

test0 = { f[a, b], f[a], f[], f, f[0, 1], f[1, 0], f[0, 0], f[1, 1], f[2, 2], f[3, 3]}

(* Paragono test e test0 *)
test0 == test

{False, f[a], f[], f, True, False, False, False, False}

```

True

```

SeedRandom[3];
tabella0 = Module[{rtemp, stemp, ftemp},
  Table[
    rtemp = RandomReal[{2, 50}];
    stemp = RandomReal[{2, 50}];
    ftemp = f[rtemp, stemp];
    {rtemp, stemp, ftemp},
    {5}]
  ];
tabella0 // TableForm

```

24.9706	2.41745	False
18.6574	8.68543	False
10.6689	27.3777	True
29.7722	38.4968	True
21.4063	45.3789	True

Ora assegno un valore alla variabile Globale x.

```

(* Ora assegno un valore alla variabile Globale x :
   questa assegnazione non ha effetto su f *)
x = 2;

```

f[99, 3]

False

```

test = { f[a, b], f[a], f[], f, f[0, 1], f[1, 0], f[0, 0], f[1, 1], f[2, 2], f[3, 3]}

{False, f[a], f[], f, True, False, False, False, False}

```

```

SeedRandom[3];
tabella = Module[{rtemp, stemp, ftemp},
  Table[
    rtemp = RandomReal[{2, 50}];
    stemp = RandomReal[{2, 50}];
    ftemp = f[rtemp, stemp];
    {rtemp, stemp, ftemp},
    {5}]
];
tabella // TableForm
(* Paragono tabella e tabella0 *)tabella0 == tabella

```

24.9706	2.41745	False
18.6574	8.68543	False
10.6689	27.3777	True
29.7722	38.4968	True
21.4063	45.3789	True

True

### □ Esercizio 2 pagina 155

Consideriamo la funzione ricorsiva **minimum** qui sotto, che trova il valore minimo in una lista .

Modificarla in modo che venga valutata SOLO se il suo argomento e' una lista di numeri.

(Suggerimento:usare List e NumberQ).

⌘ Funzione **minimum** in programmazione funzionale (usa **Which**)

**Which** valuta i vari test (uno alla volta) e restituisce il valore corrispondente al primo test ad essere verificato.

? Which

Symbol

i

Which[ $test_0, value_0, test_1, value_1, \dots$ ] evaluates each of the  $test_h$  in turn, returning  
the value of the  $value_h$  corresponding to the first one that yields True.

```

Clear[minimum, s];
(* minimum[s_?ListQ]:= *)
minimum[s_List] :=
  Which[
    (* Se s e' lista vuota, la funzione restituisce Infinity *)
    s == {}, Infinity,
    (* Se s e' un singoletto, la funzione restituisce tale valore  *)
    Length[s] == 1, s[[1]],
    (* Considero i primi due elementi della lista ed elimino
      il piu' grande dei due, prima di proseguire nella ricorsione:
      se s[[1]]>s[[2]] elimino s[[1]],
      altrimenti   elimino s[[2]] *)
    s[[1]] > s[[2]], minimum[Drop[s, {1}]],
    (* Uso True come ultimo ramo-else che combacia sempre *)
    True, minimum[Drop[s, {2}]]
    (* Drop[ list {n}]  elimina l'elemento n-esimo di list *)
  ]
TableForm[{
  minimum[{3, 5, 2, 6, 4}],
  minimum[{ }],
  (* la lista di input non e' fatta solo di numeri *)
  minimum[{a, 2}]
}]
2
∞
Which[a > 2, minimum[Drop[{a, 2}, {1}]], True, minimum[Drop[{a, 2}, {2}]]]

```

Modifico `minimum`, in modo che venga valutata SOLO se il suo argomento e' una lista di numeri.  
 Seguo il suggerimento di usare `List` e `NumberQ`.

⌘ Prima ipotesi di soluzione (usando `NumberQ`) per ottenere la modifica chiesta.  
 Nel caso in cui la lista s NON sia fatta tutta da numeri,  
 restituisco Null  
 (che non e' esattamente quanto richiesto dall'esercizio).

```
(* Applico NumberQ ad ogni elemento della lista s *)
(* minimum1[s_?ListQ]:= *)
minimum1[s_List] :=
Which[
s == {}, Infinity,
Length[s] == 1 && NumberQ[s[[1]]], s[[1]],
s[[1]] > s[[2]] && NumberQ[s[[1]]] && NumberQ[s[[2]]], minimum1[Drop[s, {1}]],
True && NumberQ[s[[1]]] && NumberQ[s[[2]]], minimum1[Drop[s, {2}]]
]
{minimum1[{3, 5, 2, 6, 4}],
minimum1[{ }],
minimum1[{a, 2}]}\!/\!\!TableForm
```

2  
 $\infty$   
Null

### ✿ Seconda ipotesi di soluzione.

Aggiungo una Condizione direttamente sull'argomento **s\_List** (tra le parentesi quadre), per controllare se/che tutta la lista **s** sia fatta da Numeri.

Realizzo tale controllo cosi':

estraggo (con Cases) dalla lista **s** una sottolista di Numeri,  
e controllo che tale sottolista coincida effettivamente con tutta la lista **s** stessa.

Se coincide, vuole dire che la lista di input e' fatta solo di numeri.

Se non coincide, vuole dire che la lista di input non e' fatta di soli numeri.

```
(* Applico NumberQ come Condition iniziale alla lista s *)
(* minimum2[ s_;/ListQ[s]&&Cases[s, _?NumberQ]==s ]:=  *)
minimum2[s_List]; Cases[s, _?NumberQ]==s]:= 
Which[
s == {}, Infinity,
Length[s] == 1, s[[1]],
s[[1]] > s[[2]], minimum2[Drop[s, {1}]],
True, minimum2[Drop[s, {2}]]
]
{minimum2[{3, 5, 2, 6, 4}],
minimum2[{ }],
minimum2[{a, 2}]//TableForm
2
∞
minimum2[{a, 2}]
```

## ? Cases

Symbol

*i*

Cases[{ $e_0, e_1, \dots$ }, pattern] gives a list of the  $e_h$  that match the pattern.

Cases[{ $e_0, \dots$ }, pattern  $\rightarrow$  rhs] gives a list of the

values of rhs corresponding to the  $e_h$  that match the pattern.

Cases[expr, pattern, levelspec] gives a list of all parts

of expr on levels specified by levelspec that match the pattern.

Cases[expr, pattern  $\rightarrow$  rhs, levelspec] gives the values of rhs that match the pattern.

Cases[expr, pattern, levelspec, n] gives the first n parts in expr that match the pattern.

Cases[pattern] represents an operator form of Cases that can be applied to an expression.

▼

□ Esercizio 3 pagina 155 (TESTO)

Scrivere una funzione `binomial[n, r]` per calcolare i coefficienti binomiali.

Fare in modo che tale funzione venga valutata solo se  $n \geq r$  ed  $r \geq 0$ .

? Binomial

Symbol

i

Binomial[ $n, m$ ] gives the binomial coefficient  $\binom{n}{m}$ .

▼

```
test = {{binomial[1, 3], 1 ≥ 3 && 3 ≥ 0}, {binomial[3, 1], 3 ≥ 1 && 1 ≥ 0},
{binomial[-3, -1], -3 ≥ -1 && -1 ≥ 0}, {binomial[-1, -3], -1 ≥ -3 && -3 ≥ 0},
{binomial[-1, 3], -1 ≥ 3 && 3 ≥ 0}, {binomial[3, -1], 3 ≥ -1 && -1 ≥ 0},
{binomial[3, 0], 3 ≥ 0 && 0 ≥ 0}, {binomial[-3, 0], -3 ≥ 0 && 0 ≥ 0}};
```

□ Esercizio 3 pagina 155 : tre soluzioni (di cui una errata)

# Immediate and Delayed Values

There are two ways to assign a value to something in *Mathematica* :

- **Set**, i.e., immediate assignment ( = )
- **SetDelayed**, i.e., delayed assignment ( := )

```
(* to define equations Equal == *)
```

In immediate assignment, the value is computed immediately when the assignment is done, and is never recomputed.

In delayed assignment, the computation of the value is delayed, and is done every time the value is requested.

For example, consider the difference between

```
valueSet = RandomColor[]
```

and

```
valueDelayed:=RandomColor[]
```

With **Set**, a random color is immediately generated :

```
SeedRandom[3];
valueSet = RandomColor[]
```

■

Every time you ask for valueSet, you get the same (random) color :

```
{valueSet, valueSet}
{■, ■}
```

With **SetDelayed**, no random color is immediately generated:

```
SeedRandom[3];
valueDelayed := RandomColor[]
```

Every time you ask for valueDelayed, **RandomColor[]** is evaluated, and a new color is generated:

```
{valueDelayed , valueDelayed }
{■, ■}

{valueSet, valueDelayed}
{■, ■}
```

It is very common to use **SetDelayed** if something is not ready yet when you are defining a value.

For example, you can make a **SetDelayed** for **Circle**, even though it does not yet have center/radius set to some value :

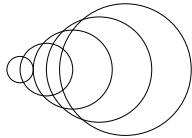
```
(* Here, n is not set *)
circles := Graphics[
  Table[
    Circle[{x, 0}, x/2],
    {x, n}
  ],
  ImageSize → Tiny
]

?circles
```

```
(* Do not ask to evaluate circles , if n is not set *)
circles
```

Once n is given a value, you can ask for circles:

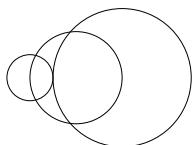
```
n = 5; circles
```



```
(* Alternatively , define circlesN as a function of n *)
circlesN[n_] := Graphics[
```

```
  Table[
    Circle[{x, 0}, x/2],
    {x, n}
  ],
  ImageSize → Tiny
]
```

```
circlesN[3]
```



```
?circlesN
```

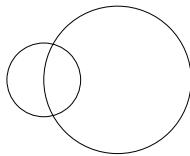
```
circlesN[]
```

```
circlesN[]
```

```
circlesN[1, 2]
```

```
circlesN[1, 2]
```

```
circlesN[{1, 2}]
(* equivalent to: Show[circlesN[1], circlesN[2]]*)
```



In delayed assignment, we do not compute a value until we need it.

There is a notion of immediate and delayed Rules too :

- **Rule**, i.e., immediate rule ( $\rightarrow$ , typed  $\rightarrow$ )
- **RuleDelayed**, i.e., delayed rule ( $\Rightarrow$ , typed  $\Rightarrow$ )

In an immediate rule  $x \rightarrow rhs$ , rhs is evaluated immediately.

In a delayed rule,  $x \Rightarrow rhs$ , rhs is evaluated every time it is requested.

For example, consider the following immediate rule, where a specific value for RandomReal[ ] is immediately computed:

```
SeedRandom[3];
x → RandomReal[];
(* Rule[x, RandomReal] *)
```

You can **Replace** four x's, but they will all be the same :

```
SeedRandom[3];
{x, x, x, x} /. x → RandomReal[]
(* ReplaceAll[ {x,x,x,x} , Rule[x, RandomReal] ] *)
{0.478554, 0.478554, 0.478554, 0.478554}
```

This is a Delayed Rule, where the computation of RandomReal[] is delayed:

```
SeedRandom[3];
x ⇒ RandomReal[];
(* RuleDelayed[x, RandomReal] *)
```

RandomReal[] is computed separately when each x is Replaced, giving four different values :

```
SeedRandom[3];
{x, x, x, x} /. x ⇒ RandomReal[]
(* ReplaceAll[ {x,x,x,x} , RuleDelayed[x, RandomReal] ] *)
{0.478554, 0.00869692, 0.347029, 0.13928}
```

## Vocabulary

$x := \text{value}$

**SetDelayed** delayed assignment, evaluated every time x is requested

$x \Rightarrow \text{value}$

**RuleDelayed** delayed rule, evaluated every time x is encountered (typed  $\Rightarrow$ )

## Timing

---

## Exercises

**39.1** Replace  $x$  in  $\{x, x+1, x+2, x^2\}$  by the same Random Integer up to 100.

**39.2** Replace each  $x$  in  $\{x, x+1, x+2, x^2\}$  by a separately chosen Random Integer up to 100.

---

## Q & A

Why not always use SetDelayed ?

What happens if I do  $x=x+1$ , with  $x$  not having a value?

Significance of inputs and outputs labeled

In[n]:=

Out[n]=

## 1.2 Ingredienti di base per un Pacchetto

Lo scopo di pacchetto è quello di fare in modo che le funzioni (che lo compongono) si comportino come le Built-in.

- Esse devono essere documentate (anche in modo da potere usare la Query ?).
- Il loro comportamento non deve dipendere da calcoli precedenti, svolti nella sessione corrente di Mathematica, prima di caricare il pacchetto.

### ■ Come (e perché) scrivere un pacchetto.

Molte sono le cose che possono andare storte:

- potremmo avere valori di variabili (usate nelle funzioni del pacchetto) già definite nella sessione;
- potremmo passare (come argomenti) variabili che sono usate anche localmente dentro le funzioni del pacchetto;
- potremmo avere dato a una funzione del pacchetto lo stesso nome di un'altra funzione già definita da qualche altra parte (shadowing);
- funzioni ausiliarie e variabili private (che sono usate dentro il pacchetto) potrebbero essere accessibili all'utente.

### ■ LINKS to Modularity and the Naming of Things

- Modules and Local Variables
  - Local Constants
  - How Modules Work
  - Variables in Pure Functions and Rules
  - Variables in Mathematics
  - Blocks and Local Values
  - Blocks Compared with Modules
  - Contexts
- **Contexts and Packages**
- Wolfram Language Packages
- **Setting Up Wolfram Language Packages**
- Files for Packages
  - Automatic Loading of Packages
  - Manipulating Symbols and Contexts by Name
  - Intercepting the Creation of New Symbols

<https://reference.wolfram.com/language/tutorial/ModularityAndTheNamingOfThings.html#3434>

### ■ LINKS to Functions and Program

- Defining Functions
  - Functions as Procedures
- **Manipulating Options**
- Repetitive Operations
  - Transformation Rules for Functions

<https://reference.wolfram.com/language/tutorial/FunctionsAndPrograms.html#15639>

### ■ LINKS to Patterns

- Introduction to Patterns
- Finding Expressions That Match a Pattern
- Naming Pieces of Patterns
- Specifying Types of Expression in Patterns

- Putting Constraints on Patterns
- Patterns Involving Alternatives
- Pattern Sequences
- Flat and Orderless Functions
- Functions with Variable Numbers of Arguments
- Optional and Default Arguments
- Setting Up Functions with Optional Arguments
- Repeated Patterns
- Verbatim Patterns
- Patterns for Some Common Types of Expression
- An Example: Defining Your Own Integration Function

<https://reference.wolfram.com/language/tutorial/Patterns.html#14984>

- ✿ Prima di vedere, per passi, un esempio di creazione di un pacchetto,  
Settiamo la Directory in cui svilupperemo il codice.

```
(* Chiediamo quale è, correntemente, la Directory di lavoro *)
currWorkDir = Directory[];

(* Chiediamo quale è la Directory di questo stesso notebook *)
noteDir = NotebookDirectory[];

(* Chiediamo quali sono i Path noti al Kernel *)
(* currWorkDir è già in $Path *)
(* noteDir non è ancora in $Path *)
$Path;
(* Part[$Path,10] *)
```

```
(* Modo 1. Inseriamo noteDir in $Path *)
(* $Path=Append[$Path, noteDir] *)
AppendTo[$Path, noteDir];
```

```
(* Modo 2. Usiamo SetDirectory in modo che noteDir diventi la Directory corrente di lavoro *)
SetDirectory[noteDir];
(* Ora la Directory corrente di lavoro è noteDir *)
Directory[] == SetDirectory[noteDir]
```

True

- ✿ Needs versus Get

Get["name.m"] legge (reads in) **name.m**,  
valuta ogni espressione in esso,

e restituisce l'ultima

(<< e' simbolo speciale di input per Get)

Se **name** e' un Contesto (i.e. **name`**), allora Get processa **name** alla ricerca del file da caricare:

il file caricato e' quello che contiene il pacchetto che definisce il Contesto **name`**

**Needs["name`"]** carica **name.m**

**Needs["context`"]** serve se il contesto specificato non e' già in \$Packages.

**Needs["context`"]** chiama **Get["context`"]**

Needs e' di norma appropriata per assicurarsi che un pacchetto sia stato caricato.

Ma se c'e' bisogno di forzare il (ri)caricamento del pacchetto, meglio Get.

→ Una situazione usuale, in cui e' meglio Get di Needs, e' durante lo sviluppo di un pacchetto, quando il programmatore cambia qualcosa nel package-file e vuole ri-caricarlo con i nuovi cambiamenti applicati.

Una funzione collegata e' **DeclarePackage**.

Get effettua un hard-load; Need fa un soft-load; DeclarePackage fa un delayed-load, ossia fa sì che il pacchetto venga caricato soltanto quando un determinato Symbol (preso da una lista avente uno o più componenti) viene usato per la prima volta.

## 1.2.0 Esempio di cattivo stile di programmazione.

Consideriamo il file **BadExample.m**, che contiene la seguente riga di codice:

```
PowerSum[x_, n_] := Sum[x^i, {i, 1, n}]
```

Carichiamo **BadExample.m**

```
(* Prima di caricare il pacchetto, quittare il kernel *)
Quit[];
(* ClearAll["Global`*"] non sarà sufficiente, perché avremo Contesti differenti dal Global *)
```

```
(* AppendTo[ $Path, NotebookDirectory[]]; *)
SetDirectory[NotebookDirectory[]];
```

```
(* Get reads in a file, evaluating each expression in it, and returning the last one *)
Get["BadExample.m"]
(* or Get["BadExample`"] or <<BadExample.m or <<BadExample` *)
```

```
(* NOTA. Dopo avere settato la Directory corretta,
possiamo usare FindFile, che aiuta appunto a trovare un file *)
(* ff=FindFile["BadExample`"];
Get[ff] *)
```

Fin qui, nessun problema; con una chiamata a **PowerSum[a, 5]** oppure a **PowerSum[x, 5]** otteniamo il risultato voluto:

```
PowerSum[a, 5]
```

$$a + a^2 + a^3 + a^4 + a^5$$

```
PowerSum[x, 5]
```

$$x + x^2 + x^3 + x^4 + x^5$$

Qui, invece, sorgono problemi.

Con una chiamata a **PowerSum[i, 5]** non otteniamo il risultato voluto.

La variabile **i** viene “catturata” dalla variabile omonima, usata (nel range {**i**, 1, **n**} della Sommatoria **Sum**) dalla funzione **PowerSum** nel pacchetto **BadExample.m**

Così, invece di ottenere  $j + j + j + j + j$ , otteniamo un numero (+ + + + +)

```
PowerSum[i, 5]
```

3413

Il problema si verifica solo con **PowerSum[i, n]**

```
PowerSum[z, 4]
```

$$z + z^2 + z^3 + z^4$$

```
?Global`*
```

▼ Global`

**i**

**n**

**PowerSum**

**x**

```
?PowerSum
```

### 1.2.1 Isoliamo le variabili locali (con Module)

Isoliamo la variabile locale **i** usata nella Sommatoria dentro il pacchetto.

Per farlo, usiamo **Module**.

In questo modo, il simbolo locale usato è sempre nuovo, e non può entrare in conflitto con nulla che venga passato come parametro(argomento) alla funzione **PowerSum[]** del pacchetto.

Consideriamo il file **BetterExample.m**, che contiene il seguente codice:

```
PowerSum[x_, n_] := Module[{i}, Sum[x^i, {i, 1, n}]]
```

(\* Prima di caricare il pacchetto, quittare il kernel \*)

```
Quit[];
```

```
SetDirectory[NotebookDirectory[]];
```

```
<< BetterExample.m
```

Nessun problema qui :

```
PowerSum[a, 5]
```

$$a + a^2 + a^3 + a^4 + a^5$$

Nessun problema anche qui :

```
PowerSum[i, 5]
```

$$i + i^2 + i^3 + i^4 + i^5$$

Resta però un problema :

i simboli ausiliari **i**, **n**, **x** (usati nel pacchetto **BetterExample.m**) sono visibili anche nel contesto **Global** (fuori dal pacchetto):

(\* notiamo la variabile "i", usata in questo notebook nella chiamata di PowerSum[i,5],  
e la sua copia locale "i\$" usata nel pacchetto BetterExample.m \*)

```
?Global`*
```

▼ Global`

**i**

**i\$**

**n**

**PowerSum**

**x**

### 1.2.2 Mettere le cose nel loro Contesto corretto(Private)

Il contesto **Context** è il meccanismo che **Mathematica** mette a disposizione, per mantenere differenti:

- le variabili usate in un pacchetto,
- dalle variabili usate nella sessione principale.

Ogni simbolo appartiene ad un determinato Contesto.

All'interno di un unico Contesto, i nomi dei simboli sono univoci.

Uno stesso nome, invece, può apparire in due Contesti differenti.

Di default, tutti i nuovi simboli che noi definiamo vengono messi nel contesto **Global`**.

In **BetterExample.m** perfino **i\$** (ossia la variabile i locale) entra nel contesto **Global`**.

Per evitare ciò, dobbiamo dire a **Mathematica** di creare nuovi simboli in un contesto differente.

Consideriamo il file **BestExample.m**, che contiene il seguente codice:

```
PowerSum::usage= "PowerSum[x_, n_] returns the sum of the first n powers of x."
```

```
Begin["Private`"]
```

```
PowerSum[x_, n_]:= Module[{i}, Sum[x^i, {i, 1, n}]]
```

```
End[]
```

(\* Prima di caricare il pacchetto , quitare il kernel \*)

```
Quit[];
```

```
SetDirectory[NotebookDirectory[]];
<< BestExample.m;
```

Questa volta, la variabile i locale (i\$) viene creata dentro al contesto Private`, che **non** verrà esaminato quando (successivamente) noi useremo nomi di variabile nella sessione corrente.

Il simbolo **PowerSum** (che è il nome della funzione che vogliamo usare, dopo avere caricato il pacchetto che la contiene), invece, dove ovviamente essere visibile nel contesto Global`.

Nel pacchetto **BestExample.m** inseriamo (fuori del contesto Private) la riga di documentazione (**usage**) della funzione **PowerSum[]**.

```
PowerSum[a, 5]
PowerSum[i, 5]
```

$$a + a^2 + a^3 + a^4 + a^5$$

$$i + i^2 + i^3 + i^4 + i^5$$

?Global`\*

Symbol
PowerSum[x, n] returns the sum of the first n powers of x.

▼ Global`

a	i	PowerSum
---	---	----------

(\* i, n, x sono simboli locali, dovuti a Sum e PowerSum \*)

(\* i\$ e' copia locale della variabile globale i \*)

?Private`\*

▼ Private`

i	i\$	n	x
---	-----	---	---

?PowerSum

Symbol
PowerSum[x, n] returns the sum of the first n powers of x.

### 1.2.3 Il contesto di un pacchetto

Oltre a nascondere le variabili locali e le funzioni ausiliarie (in modo che non siano visibili nel contesto Globale),

vogliamo mettere tutte le funzioni del pacchetto in un unico contesto separato (e con un proprio nome, che non sia il generico Private`).

Tale nome (da noi scelto per il contesto del pacchetto), però, deve essere visibile (altrimenti non potremmo usarne le funzionalità interne).

Questo si ottiene con **BeginPackage[] / EndPackage[]**

Consideriamo un nuovo pacchetto esemplificatore: **MappaCartesiana.m**

che contiene il codice che segue, e che crea il grafico di una griglia del piano cartesiano, in cui i vertici sono punti {x, y} trasformati da una funzione f

Nota.

Prima vengono formati i valori  $t = f[x + I y]$ :

poi ciascun valore t viene separato nelle sue parti Reale e Immaginaria.

Le coppie {Re[t], Im[t]} vengono infine graficate in forma parametrica:

- i valori orizzontali (Table con spread = {x, x0, x1, dx}) dipendono solo da y , i.e. le coppie sono di tipo {Re[y], Im[y]} e ParametricPlot bounds={y, y0, y1};
- i valori verticali (Table con spread = {y, y0, y1, dy}) dipendono solo da x , i.e. le coppie sono di tipo {Re[x], Im[x]} e ParametricPlot bounds={x, x0, x1}.

**CODICE** di **MappaCartesiana.m**

Notiamo che in **Begin["`Private`"]** c'è un doppio "tick".

Questo indica che il contesto **Private** è un sotto-contesto del contesto **MappaCartesiana`** del pacchetto.

```
(* Prima di caricare il pacchetto, quitare il kernel *)
```

```
Quit[];
```

```
SetDirectory[NotebookDirectory[]];
```

```
<< MappaCartesiana.m
```

La funzione **CartesianMap** è nel proprio contesto **MappaCartesiana`**

```
Context[CartesianMap]
```

**MappaCartesiana`**

Il contesto **MappaCartesiana`** è accessibile, perché è stato aggiunto al Path per la ricerca di Contesti:

```
$ContextPath
```

```
{DocumentationSearch`, ResourceLocator`, MappaCartesiana`, System`, Global`}
```

Chiamo la funzione:

```
?CartesianMap
```

Symbol

CartesianMap[ f , {x0, x1, dx}, {y0, y1, dy}] plots the image  
of Cartesian coordinate lines, modified under the action of function f .



```
CartesianMap[Sin, {-1, 1, 0.5}, {-1, 1, 0.5}]
```

```
CartesianMap[Sin, {-1, 1, 0.2}, {-2, 2, 0.2}]
```

```
CartesianMap[Exp, {-1, 1, 0.2}, {-2, 2, 0.2}]
```

```
CartesianMap[Cos, {0.2, Pi - 0.2, (Pi - 0.4)/19}, {-2, 2, 4/16}]
```

La funzione ausiliaria **Curves** sta nel contesto **Privato** del pacchetto **MappaCartesiana**, quindi **non** e' accessibile da fuori.

```
?Curves
```

```
Missing[UnknownSymbol, Curves]
```

```
Context[Curves]
```

```
Global`
```

```
3 (* :Title: BadExample *)
4 (* :Context: BadExample` *)
5 (* :Author: GS *)
6 (* :Summary: a function with a local variable that has global scope      *)
7 (* :Copyright: GS 2023 *)
8 (* :Package Version: 1 *)
9 (* :Mathematica Version: 13 *)
10 (* :History: last modified 27/3/2023 *)
11 (* :Keywords: scope, nesting, bad programming style *)
12 (* :Sources: book / link *)
13 (* :Limitations: this is a preliminary version, for educational purposes only. *)
14 (* :Discussion: *)
15 (* :Requirements: *)
16 (* :Warnings: as the name implies, this is an example of bad programming *)
17
18 (* This function returns the sum of the first n powers of x *)
19 PowerSum[x_, n_] := Sum[ x^i, {i, 1, n} ]
```

```

3 (* :Title: BestExample *)
4 (* :Context: BestExample` *)
5 (* :Author: GS *)
6 (* :Summary: an example of good programming style *)
7 (* :Copyright: GS 2023 *)
8 (* :Package Version: 1 *)
9 (* :Mathematica Version: 13 *)
10 (* :History: last modified 27/3/2023 *)
11 (* :Keywords: programming style, local variables *)
12 (* :Sources: biblio *)
13 (* :Limitations: this is for educational purposes only. *)
14 (* :Discussion: *)
15 (* :Requirements: *)
16 (* :Warning: package Context is not defined *)
17
18 PowerSum::usage = "PowerSum[x, n] returns the sum of the first n powers of x."
19
20 Begin["Private`"]
21
22 PowerSum[x_, n_] :=
23   Module[{i},
24     Sum[x^i, {i, 1, n}]
25   ]
26
27 End[]

```

```
3 (* :Title: BetterExample *)
4 (* :Context: BetterExample` *)
5 (* :Author: GS *)
6 (* :Summary: an example of poor programming style. *)
7 (* :Copyright: GS 2023 *)
8 (* :Package Version: 1 *)
9 (* :Mathematica Version: 13 *)
10 (* :History: last modified 27/3/2023 *)
11 (* :Keywords: template, skeleton, package *)
12 (* :Sources: biblio *)
13 (* :Limitations:
14     this is a preliminary version, for educational purposes only. *)
15 (* :Discussion: *)
16 (* :Requirements: *)
17 (* Warning : i simboli i, n, x sono visibili nel contesto Globale *)
18
19 (* PowerSum::usage = "PowerSum[x, n] returns the sum of the first n powers of x." *)
20
21 PowerSum[x_, n_] :=
22     Module[{i},
23         Sum[x^i, {i, 1, n}]
24     ]
25
26
```

```

3 (* :Title: MappaCartesiana *)
4 (* :Context: MappaCartesiana` *)
5 (* :Author: GS *)
6 (* :Summary: a preliminary version of the ComplexMap package *)
7 (* :Copyright: GS 2023 *)
8 (* :Package Version: 2 *)
9 (* :Mathematica Version: 13 *)
10 (* :History: last modified 27/3/2023 *)
11 (* :Keywords: programming style, local variables *)
12 (* :Sources: biblio *)
13 (* :Limitations:
14     this is a preliminary version, for educational purposes only. *)
15 (* :Discussion: *)
16 (* :Requirements: *)
17 (* :Warning: DOCUMENTARE TUTTO il codice *)

20 BeginPackage["MappaCartesiana`"]
23 CartesianMap::usage =
24 "CartesianMap[ f , {x0, x1, dx}, {y0, y1, dy}] plots the image of Cartesian
25 coordinate lines, modified under the action of function f ."
28 Begin["`Private`"]

```

```

31 CartesianMap[ func_, {x0_, x1_, dx_}, {y0_, y1_, dy_} ] :=
32   Module[ {w, x, y, horizontalgrid, verticalgrid},
33
34     (* Use func and the coordinates {x,y} to form the value f[x+I*y] *)
35
36     w = func[x + I y];
37
38     (* Form the table of horizontal values, with spread={x, x0, x1, dx}.
39      Horizontal coordinates turn out to be of the type { Re[y] , Im[y] }
40      thus ParametricPlot (called by Curves[]) has bounds={y, y0, y1} *)
41
42     horizontalgrid= Curves[ w, {x, x0, x1, dx}, {y, y0, y1} ];
43
44     (* Form the table of vertical values, with spread={y, y0, y1, dy}.
45      Vertical coordinates turn out to be of the type { Re[x] , Im[x] }
46      thus ParametricPlot (called by Curves[]) has bounds={x, x0, x1} *)
47
48     verticalgrid = Curves[ w, {y, y0, y1, dy}, {x, x0, x1} ];
49
50     Show[ Graphics[ Join[horizontalgrid, verticalgrid] ],
51           AspectRatio→Automatic,
52           Axes→True ]
53   ]

```

```
56 Curves[w_, spread_, bounds_] :=
57 Module[{curves},
58   curves = Table[ { Re[w], Im[w] } , spread ];
59
60 (* ParametricPlot[ { fx , fy } , { t, tmin, tmax} ]
61 plots a curve in parametric form,
62 i.e. { x==fx[t] , y==fy[t] },
63 that is a curve with coordinates {x, y} given by {fx, fy} ,
64 where fx = fx[t] , fy = fy[t] *)
65
66 ParametricPlot[ curves, bounds ]][1]
67 ]
68
69 End[]
70
71 EndPackage[]
```

### 1.3 Aggiungere un'altra funzione al Pacchetto

Aggiungiamo una funzione per fare il Plot non solo in coordinate cartesiane  $\{x, y\}$ , ma anche in coordinate polari  $\{r, \phi\}$ .

I numeri complessi  $x + iy$  diventano  $r * \text{Exp}[I \phi]$ .

Consideriamo il file **MappaComplessa0.m**, che contiene il seguente CODICE:

Le funzioni ausiliarie **Picture** e **Curves** sono nel contesto **Privato** del pacchetto **MappaComplessa0**, quindi **non** sono accessibili da fuori.

Invece **CartesianMap** e **PolarMap** (attraverso la `usage::`) sono accessibili da fuori.

```
(* Eventuale Quit del Kernel *)
```

```
Quit[];
```

```
(* AppendTo[ $Path, NotebookDirectory[]]; *)
```

```
SetDirectory[NotebookDirectory[]];
```

```
<< MappaComplessa0.m
```

```
?PolarMap
```

Symbol

**PolarMap**[ $f$ ,  $\{r_0, r_1, dr\}$ ,  $\{p_0, p_1, dp\}$ ] plots the image

of the polar coordinate lines under the function  $f$ .

```
?CartesianMap
```

Symbol

**CartesianMap**[ $f$ ,  $\{x_0, x_1, dx\}$ ,  $\{y_0, y_1, dy\}$ ] plots the image

of the Cartesian coordinate lines under the function  $f$ .

```
?Picture
```

```
Missing[UnknownSymbol, Picture]
```

```
?Curves
```

```
Missing[UnknownSymbol, Curves]
```

```
(* Il contesto MappaComplessa0` e' stato aggiunto a $ContextPath,
```

```
quando abbiamo usato Get per caricare il pacchetto MappaComplessa0` *)
```

```
$ContextPath
```

```
{MappaComplessa0`, DocumentationSearch`, ResourceLocator`, System`, Global`}
```

```
(* MappaComplessaO è il nome dato al pacchetto/libreria di programmi ,
specificato come argomento di BeginPackage[ ] *)
(* BeginPackage[ "MappaComplessaO`" ] ..... EndPackage[ ] *)

(* MappaComplessaO e' nel contesto Global` *)
Context[MappaComplessaO]
```

Global`

```
(* PolarMap e CartesianMap sono main-routine nel pacchetto MappaComplessaO .
Stanno dopo BeginPackage e prima di BeginPrivate *)
(* PolarMap e CartesianMap sono nel contesto MappaComplessaO` *)
{Context[PolarMap], Context[CartesianMap]}
```

{MappaComplessaO`, MappaComplessaO`}

```
(* Picture e' ausiliaria a PolarMap e CartesianMap ,
Curves e' ausiliaria a Picture.*)
(* Picture e Curves stanno nel Private di MappaComplessaO *)
(* Quindi , fuori dal pacchetto MappaComplessaO ,
i nomi (blu) Picture e Curves sono visti come potenziali nomi nel contesto Global` *)
{Context[Picture], Context[Curves]}
```

{Global`, Global`}

Immagine delle linee delle coordinate polari in scala logaritmica(che è inversa della esponenziale).

```
PolarMap[Log, {0.1, 10, 0.5}, {-3, 3, 0.15}]
```

```
CartesianMap[Log, {0.1, 10, 0.5}, {-3, 3, 0.15}]
```

Chiamo la funzione, dando come primo argomento una funzione pura:

```
PolarMap[
 1/Conjugate[##] &,
 {0.1, 5.1, 0.5}, {-Pi, Pi, Pi/24}
]
```

```
PolarMap[
 Identity,
 {1, 2, 0.1}, {-Pi, Pi, Pi/12}
]
```

```

3 (* :Title: MappaComplessa0 *)
4 (* :Context: MappaComplessa0` *)
5 (* :Author: GS *)
6 (* :Summary: rispetto a MappaCartesiana.m,
7     qui viene aggiunta una funzione per plottare
8     non solo in coordinate cartesiane, ma anche polari *)
9 (* :Copyright: GS 2023 *)
10 (* :Package Version: 2 *)
11 (* :Mathematica Version: 13 *)
12 (* :History: last modified 27/3/2023 *)
13 (* :Keywords: Cartesian coordinates, polar coordinates *)
14 (* :Sources: biblio *)
15 (* :Limitations:
16     this is a preliminary version, for educational purposes only. *)
17 (* :Discussion: *)
18 (* :Requirements: *)
19 (* :Warning: DOCUMENTATE TUTTO il codice *)

22 BeginPackage["MappaComplessa0`"]
25
26 CartesianMap::usage =
27 "CartesianMap[f, {x0, x1, dx}, {y0, y1, dy}] plots the image
28 of the Cartesian coordinate lines under the function f."
30
31 PolarMap::usage =
32 "PolarMap[f, {r0, r1, dr}, {p0, p1, dp}] plots the image
33 of the polar coordinate lines under the function f."
35
36 Begin["`Private`"]
38
39 CartesianMap[ func_, {x0_, x1_, dx_}, {y0_, y1_, dy_} ] :=
40
41     Module[ {x, y},
42
43         (* Use func and the cartesian coordinates {x,y} to form the value f[x+I*y] *)
44
45             Picture[ func[x + I y], {x, x0, x1, dx}, {y, y0, y1, dy} ]
46     ]

```

```

47 PolarMap[ func_, {r0_, r1_, dr_}, {p0_, p1_, dp_} ] :=
48   Module[ {r, p},
49
50     (* Use func and the polar coordinates {r,p} to form the value f[r*Exp[I*p]] *)
51
52     Picture[ func[r Exp[I p]], {r, r0, r1, dr}, {p, p0, p1, dp} ]
53   ]
54
55
56 Picture[ v_, {s_, s0_, s1_, ds_}, {t_, t0_, t1_, dt_} ] :=
57   Module[ {horiz, vert},
58
59     (* Form the table of horizontal values, with spread={s, s0, s1, ds}.
60       Horizontal coordinates are of the type { Re[y] , Im[y] } or { Re[p] , Im[p] }
61       thus ParametricPlot (called by Curves[]) has bounds={t, t0, t1} *)
62
63     horiz = Curves[ v, {s, s0, s1, ds}, {t, t0, t1} ];
64
65     (* Form the table of vertical values, with spread={t, t0, t1, dt}.
66       Vertical coordinates are of the type { Re[x] , Im[x] } or { Re[r] , Im[r] }
67       thus ParametricPlot (called by Curves[]) has bounds={s, s0, s1} *)
68
69     vert = Curves[ v, {t, t0, t1, dt}, {s, s0, s1} ];
70
71     Show[ Graphics[ Join[horiz, vert] ],
72           AspectRatio→Automatic, Axes→True ]
73   ]

```

```
76 Curves[w_, spread_, bounds_] :=
77 Module[{curves},
78   curves = Table[{Re[w], Im[w]}, {spread}] ;
79
80   (* ParametricPlot[ {fx, fy}, {u, umin, umax}] *)
81   plots a curve in parametric form,
82   i.e. {x==fx[u], y==fy[u]},
83   that is a curve with coordinates {x, y} given by {fx, fy} ,
84   where fx = fx[u], fy = fy[u] *)
85
86   ParametricPlot[curves, bounds][[1]]
87 ]
88
89 End[ ]
90
91 EndPackage[ ]
```

## 1.4 Aggiungere Defaults e Options

In entrambe le funzioni **CartesianMap** e **PolarMap**, il range per le variabili

(rispettivamente cartesiane o polari) è dato come lista di 3 elementi

{ start , final , increment }

È utile poter dare ad increment un valore di default

(come accade in altri costrutti con indice iteratore, come **Table**, per esempio).

Nel fare ciò, però, non vogliamo dare all'incremento un numero fisso (come default fisso).

Vogliamo invece poter specificare un numero **n** qualsiasi di punti/linee,

da graficare tra **start** e **final**, in **CartesianMap** o **PolarMap**,

in base al quale il default di **increment** venga calcolato come **(final - start)/(n-1)**.

In altre parole, vogliamo dare un valore **n** che non è fisso.

Una buona scelta è usare il valore (di default) della Option **PlotPoints** di **Plot**

(dato che anche **PlotPoints** calcola il numero di punti/linee che **Plot** deve disegnare).

Valori di default che devono essere calcolati (come quello per **increment**)

non possono essere dati dentro il pattern (e.g. increment\_) nella definizione di una funzione.

Dobbiamo, quindi, dare due Regole di definizione della nostra funzione,

in modo che la seconda combaci quando viene omesso, in chiamata, il parametro di cui stiamo definendo un default

(mentre la prima regola è quella già vista in **MappaComplessa0**, che combacia quando tutti i parametri di chiamata vengono dati)

→ In sintesi, nella **CartesianMap** (e analogamente nella **PolarMap**) :

**1a.** diamo una definizione di funzione

con due soli argomenti in ascissa { **x0** , **x1** } e in ordinata { **y0** , **y1** } ,

senza specificare i pattern **dx\_** e **dy\_** (per l'incremento di ascisse e ordinate, rispettivamente);

**1b.** poi, nel body della funzione, definiamo una Regola che servirà ad assegnare un valore a **dx** e **dy** ;

**1c.** calcoliamo **dx** e **dy** ;

**1d.** infine, chiamiamo (ricorsivamente) **CartesianMap**

con tre argomenti in ascissa { **x0** , **x1** , **dx** } e in ordinata { **y0** , **y1** , **dy** }).

**2.** A questo punto, la regola data precedentemente per **CartesianMap** combacerà sugli argomenti e il disegno verrà graficato.

Modifichiamo **MappaComplessa0.m** (e salviamolo come **MappaComplessa2.m**) in modo che contenga il seguente CODICE:

Per **PolarMap** faremo una modifica analoga.

Dato che, quasi sempre, il raggio iniziale partirà dall'origine,

possiamo anche aggiungere un valore di default ad **r0** , scrivendo **r0\_:0**

(questo è un caso più semplice di quello di prima,

dato che stavolta vogliamo semplicemente specificare un numero costante, di default, per **r0** ).

Consideriamo il file **MappaComplessa2.m** che contiene il codice visto.

Notiamo anche l'aggiornamento del messaggio di **usage**.

(\* Prima, eseguire il Quit del Kernel \*)  
 Quit[]

SetDirectory[ NotebookDirectory[ ]];  
 << MappaComplessa2.m

?PolarMap

#### Symbol

PolarMap[f, {r0:0, r1, (dr)}, {p0, p1, (dp)}]

plots the image of the polar coordinate lines under the function f.

The default values of dr and dphi are chosen so that the number of lines  
 is equal to the value of the option PlotPoints of Plot3D[ ].

?CartesianMap

#### Symbol

CartesianMap[f, {x0, x1, (dx)}, {y0, y1, (dy)}]

plots the image of the cartesian coordinate lines under the function f.

The default values of dx and dy are chosen so that the number of lines  
 is equal to the value of the option PlotPoints of Plot3D[ ].

(\* Le funzioni ausiliarie sono nel contesto Privato \*)

?Picture

?Curves

Missing[UnknownSymbol, Picture]

Missing[UnknownSymbol, Curves]

(\* MappaComplessa2` e in \$ContextPath \*)

\$ContextPath

{DocumentationSearch`, ResourceLocator`, MappaComplessa2`, System`, Global`}

(\* Il Context di MappaComplessa2, Picture, Curves e' Global` \*)

(\* Il Context di PolarMap, CartesianMap e' MappaComplessa2` \*)

Map[Context, {MappaComplessa2, Picture, Curves, PolarMap, CartesianMap}]

{Global`, Global`, Global`, MappaComplessa2`, MappaComplessa2`}

**Nota su PlotPoints/.Options[Plot]**

```
(* Alteriamo il valore di PlotPoints *)
SetOptions[ Plot, PlotPoints → 7];
Timing[CartesianMap[Zeta, {0.1, 0.9}, {0, 20}]]
```

```
SetOptions[ Plot, PlotPoints → 15];
Timing[CartesianMap[Zeta, {0.1, 0.9}, {0, 20}]]
```

Immagine delle linee delle coordinate polari in scala logaritmica (che è inversa della esponenziale).

```
SetOptions[ Plot, PlotPoints → 7];
Timing[PolarMap[Sqrt, {1}, {-Pi + 0.0001, Pi}]]
```

```
SetOptions[ Plot, PlotPoints → 15];
Timing[PolarMap[Sqrt, {1}, {-Pi + 0.0001, Pi}]]
```

**NOTA.** Riportiamo il valore di PlotPoints ad Automatic:

```
SetOptions[ Plot, PlotPoints → Automatic];
```

(\*\*\*\*\*)

Possiamo considerare la variante [MappaComplessa2MakeLines.m](#)  
 Essa contiene una riscrittura delle funzioni ausiliarie, compattate in una sola,  
 che lavora in MachinePrecision ed usa **Line**, invece che **ParametricPlot**  
 (quindi è più veloce, ma più grossolana), ed ha una conseguente resa in **Bianco/Nero** dei grafici.

```
(* Prima, eseguire il Quit del Kernel ;
Quit[] quits only the kernel, not the front end. *)
Quit[]
```

```
SetDirectory[NotebookDirectory[]];
<< MappaComplessa2MakeLines.m
```

```
(* La funzione ausiliaria sta nel contesto Privato *)
? MakeLines
```

```
Missing[UnknownSymbol, MakeLines]
```

```
(* Il Context di MappaComplessa2MakeLines, MakeLines, e' Global` *)
(* Il Context di PolarMap, CartesianMap e' MappaComplessa2` *)
Map[Context, {MappaComplessa2MakeLines, MakeLines, PolarMap, CartesianMap}]
```

```
{Global`, Global`, MappaComplessa2MakeLines`, MappaComplessa2MakeLines`}
```

```
SetOptions[ Plot, PlotPoints → 10];
Timing[CartesianMap[Zeta, {0.1, 0.9}, {0, 20}]]
```

```
SetOptions[ Plot, PlotPoints → 50];
Timing[CartesianMap[Zeta, {0.1, 0.9}, {0, 20}]]
```

```
SetOptions[ Plot, PlotPoints → 50];
Timing[PolarMap[Sqrt, {1}, {-Pi + 0.0001, Pi}]]
```

(\*\*\*\*\*)

```
SetOptions[ Plot, PlotPoints → Automatic];
```

```
3 (* :Title: MappaComplessa2 *)
4 (* :Context: MappaComplessa2` *)
5 (* :Author: GS *)
6 (* :Summary: rispetto a MappaComplessa0.m,
7     qui viene aggiunta una seconda Rule
8     per dare valori di default agli incrementi dx , dy *)
9 (* :Copyright: GS 2023 *)
10 (* :Package Version: 2 *)
11 (* :Mathematica Version: 13 *)
12 (* :History: last modified 27/3/2023 *)
13 (* :Keywords: default values *)
14 (* :Sources: biblio *)
15 (* :Limitations:
16     this is a preliminary version, for educational purposes only. *)
17 (* :Discussion: *)
18 (* :Requirements: *)
19 (* :Warning: DOCUMENTATE TUTTO il codice *)
```

```
22 BeginPackage["MappaComplessa2`"]
```

```
25 CartesianMap::usage = "CartesianMap[f, {x0, x1, (dx)}, {y0, y1, (dy)}]
26     plots the image of the cartesian coordinate lines under the function f.
27     The default values of dx and dy are chosen so that the number of lines
28     is equal to the value of the option PlotPoints of Plot3D[ ]."
```

```
31 PolarMap::usage = "PolarMap[f, {r0:0, r1, (dr)}, {p0, p1, (dp)}]
32     plots the image of the polar coordinate lines under the function f.
33     The default values of dr and dphi are chosen so that the number of lines
34     is equal to the value of the option PlotPoints of Plot3D[ ]."
```

```
37 Begin["`Private`"]

```

```
40 (* default increments *)
41 CartesianMap[ func_ , {x0_, x1_}, {y0_, y1_} ] :=
42   Module[ {dx, dy, plotpoints},
43
44   (* Di default, PlotPoints/.Options[Plot] e' Automatic *)
45   (* La Option PlotPoints puo' essere modificata con SetOptions[ Plot, PlotPo-
46   (* Qui, pertanto, colleghiamo plotpoints ad una Option di Plot il cui default
47
48     plotpoints = PlotPoints /. Options[Plot];
49     Print["current value of plotpoints= ",plotpoints];
50
51     dx=(x1-x0)/(plotpoints-1);
52     dy=(y1-y0)/(plotpoints-1);
53   CartesianMap[ func, {x0, x1, dx}, {y0, y1, dy} ]
54   ]
55
56 (* explicit increments *)
57 CartesianMap[ func_, {x0_, x1_, dx_}, {y0_, y1_, dy_} ] :=
58   Module[ {x, y},
59     Picture[ func[x + I y], {x, x0, x1, dx}, {y, y0, y1, dy} ]
60   ]
```

```

63 (* default increments *)
64 PolarMap[ func_, {r0_:0, r1_}, {p0_, p1_} ] :=
65   Module[ {dr, dp, plotpoints},
66
67     (* Idem *)
68     (* Qui colleghiamo plotpoints ad una Option di Plot il cui default puo' essere
69      plotpoints = PlotPoints /. Options[Plot];
70
71      dr=(r1-r0)/(plotpoints-1);
72      dp=(p1-p0)/(plotpoints-1);
73
74    PolarMap[ func, {r0, r1, dr}, {p0, p1, dp} ]
75  ]
76
77 (* explicit increments *)
78 PolarMap[ func_, {r0_, r1_, dr_}, {p0_, p1_, dp_} ] :=
79   Module[ {r, p},
80     Picture[ func[r Exp[I p]], {r, r0, r1, dr}, {p, p0, p1, dp} ]
81   ]
82
83
84 (* auxiliary functions *)
85 Picture[ v_, {s_, s0_, s1_, ds_}, {t_, t0_, t1_, dt_} ] :=
86   Module[ {hg, vg},
87     hg = Curves[ v, {s, s0, s1, ds}, {t, t0, t1} ];
88     vg = Curves[ v, {t, t0, t1, dt}, {s, s0, s1} ];
89     Show[ Graphics[ Join[hg, vg] ],
90           AspectRatio->Automatic, Axes->True ]
91   ]
92
93 Curves[ w_, spread_, bounds_ ] :=
94   Module[ {curves} ,
95     curves = Table[ { Re[w] , Im[w] } , spread ] ;
96     ParametricPlot[curves, bounds][[1]]
97   ]
98
99
100 End[ ]

```

103 EndPackage[ ]

```

3 (* :Title: MappaComplessa2MakeLines *)
4 (* :Context: MappaComplessa2MakeLines`)
5 (* :Author: GS *)
6 (* :Summary: variante di MappaComplessa2, più veloce, ma più grossolana;
7     qui lavoriamo in MachinePrecision, in B/W invece che a colori,
8     ed usiamo Line invece che Parametric Plot *)
9 (* :Copyright: GS 2023 *)
10 (* :Package Version: 2 *)
11 (* :Mathematica Version: 13 *)
12 (* :History: last modified 27/3/2023 *)
13 (* :Sources: biblio *)
14 (* :Limitations:
15     this is a preliminary version, for educational purposes only. *)
16 (* :Discussion: USES LINE *)
17 (* :Requirements: *)
18 (* :Warning: DOCUMENTATE TUTTO il codice *)

21 BeginPackage["MappaComplessa2MakeLines`"]

24 CartesianMap::usage = "CartesianMap[f, {x0, x1, (dx)}, {y0, y1, (dy)}]
25     plots the image of the cartesian coordinate lines under the function f.
26     The default values of dx and dy are chosen so that the number of lines
27     is equal to the value of the option PlotPoints of Plot3D[ ]."

30 PolarMap::usage = "PolarMap[f, {r0:0, r1, (dr)}, {p0, p1, (dp)}]
31     plots the image of the polar coordinate lines under the function f.
32     The default values of dr and dphi are chosen so that the number of lines
33     is equal to the value of the option PlotPoints of Plot3D[ ]."

36 Begin["`Private`"]

```

```

39 (* default increments *)
40 CartesianMap[ func_ , {x0_, x1_}, {y0_, y1_} ] :=
41   Module[ {dx, dy, plotpoints},
42
43   (* Di default, PlotPoints/.Options[Plot] e' Automatic *)
44   (* La Option PlotPoints puo' essere modificata con SetOptions[ Plot, PlotPo-
45   (* Qui, pertanto, colleghiamo plotpoints ad una Option di Plot il cui default
46
47     plotpoints = PlotPoints /. Options[Plot];
48     (* Print["current value of plotpoints= ",plotpoints];    *)
49
50     dx=(x1-x0)/(plotpoints-1);
51     dy=(y1-y0)/(plotpoints-1);
52
53   CartesianMap[ func, {x0, x1, dx}, {y0, y1, dy} ]
54   ]
55
56 (* explicit increments *)
57 CartesianMap[ func_, {x0_, x1_, dx_}, {y0_, y1_, dy_} ] :=
58   Module[ {x, y, coords},
59     coords = Table[ N[ func[x + I y] ] , {x, x0, x1, dx}, {y, y0, y1, dy} ];
60     Show[ MakeLines[coords], AspectRatio -> Automatic, Axes -> Automatic]
61   ]

```

```

64 (* default increments *)
65 PolarMap[ func_, {r0_:0, r1_}, {p0_, p1_} ] :=
66   Module[ {dr, dp, plotpoints},
67
68     (* Idem *)
69     (* Qui colleghiamo plotpoints ad una Option di Plot il cui default puo'
70       plotpoints = PlotPoints /. Options[Plot];
71
72       dr=(r1-r0)/(plotpoints-1);
73       dp=(p1-p0)/(plotpoints-1);
74
75   PolarMap[ func, {r0, r1, dr}, {p0, p1, dp} ]
76   ]
77
78 (* explicit increments *)
79 PolarMap[ func_, {r0_, r1_, dr_}, {p0_, p1_, dp_} ] :=
80   Module[ {r, p, coords},
81     coords = Table[ N[ func[r Exp[I p]] ], {r, r0, r1, dr}, {p, p0, p1, dp} ];
82     Show[ MakeLines[coords], AspectRatio -> Automatic, Axes -> Automatic]
83   ]
84
85 (* auxiliary function *)
86 MakeLines[points_] :=
87   Module[ {coords, lines},
88     coords = Map[ {Re[#], Im[#]} &, points, {2} ];
89     lines = Map[ Line, Join[ coords, Transpose[coords] ] ];
90     Graphics[ lines ]
91   ]
92
93 End[ ]
94
95 EndPackage[ ]

```

## 1.5 Un altro modo di aggiungere Defaults

Abbiamo visto come dare un valore di default ad entrambi gli **dy** e **dr, dp**, rispettivamente delle funzioni **CartesianMap** e **PolarMap**.

Vediamo ora come dare ai due incrementi valori di default slegati l'uno dall'altro.

In particolare, vediamo come effettuare chiamate “miste” (con un incremento dato esplicitamente e l'altro implicitamente): ad esempio, **PolarMap[ Sqrt, { 0, 1, 0.2 }, { -Pi - 0.0001, Pi } ]**.

Per come è stata definita **CartesianMap** fin qui (e analogamente per **PolarMap**), una chiamata mista non verrebbe valutata, perché non c'è una regola che la preveda

(c'è una regola per **CartesianMap** con gli incrementi dati entrambi esplicitamente o entrambi implicitamente; il caso misto non è stato ancora definito).

Potremmo aggiungere altre due regole che prevedano i due casi misti:  
**{ dx esplicito, dy implicito }** e **{ dx implicito, dy esplicito }**.

Vediamo un modo migliore.

L'idea è :

- usare un qualche simbolo come valore di default per l'incremento;
- poi, testare la presenza (o meno) di tale simbolo.

Possiamo usare come simbolo **Automatic**,

che viene già usato come valore di default in molte Options di Built-in (di grafica, e non solo di grafica).

Per testare se il valore dell' incremento coincide con tale simbolo, si usa **SameQ[e1, e2]**,

che controlla la uguaglianza letterale tra FullForm dei simboli **e1, e2** (a differenza di **Equal[e1, e2]**, che testa la uguaglianza numerica tra **e1, e2**, restituendo True/False se sono uguali/diversi, oppure Unevaluated se la uguaglianza non puo' essere stabilita, come nel caso di Equal[1, a] con a non assegnato).

(\* Un esempio, per ricordare come funzionano Equal e SameQ \*)

```
{Sqrt[2] + Sqrt[3] == Sqrt[5 + 2 Sqrt[6]],  
 Sqrt[2] + Sqrt[3] === Sqrt[5 + 2 Sqrt[6]]}
```

{True, False}

Usando Automatic e SameQ, possiamo unificare le due regole che avevamo dato per **CartesianMap**  
(e analogamente per **PolarMap**).

Consideriamo il file **MappaComplessa3.m** che contiene il seguente codice  
(oltre alla funzione ausiliaria **MakeLines**):

```

CartesianMap[ func_,
  {x0_, x1_, dx_:Automatic}, {y0_, y1_, dy_:Automatic}] :=
Module[ {x, y, coords, plotpoints, ndx = dx, ndy = dy},
  (* L' incremento locale ndx è inizializzato al valore dx dato in input *)
  (* Idem per dy *)

  (* plotpoints è settata al valore PlotPoints della relativa
   Option di Plot, che possiamo modificare con SetOptions *)
  plotpoints = PlotPoints /. Options[Plot];

  (* Se SameQ[dx, Automatic] restituisce True,
   vuole dire che dx non è stato dato esplicitamente
   nella chiamata a CartesianMap:
   in questo caso, usiamo plotpoints per calcolare ndx *)
  (* Idem per dy *)
  If[ dx === Automatic, ndx = N[(x1 - x0) / (plotpoints - 1)];]
  If[ dy === Automatic, ndy = N[(y1 - y0) / (plotpoints - 1)];]

  (* Arrivati qui, ndx ha il valore dx dato esplicitamente in
   chiamata oppure il valore calcolato attraverso plotpoints *)
  (* Idem per dy *)
  coords = Table[ N[ func[ x + l y]], {x, x0, x1, ndx}, {y, y0, y1, ndy}];

  (* Chiamata alla routine ausiliaria MakeLines *)
  Show[MakeLines[coords],
    AspectRatio → Automatic, Axes → Automatic]
]

```

Per PolarMap faremo una modifica analoga, mantenendo inoltre il valore zero

come default per `r0`, ossia `r0_:0`.

(\* Quit del Kernel \*)

`Quit[]`

`SetDirectory[NotebookDirectory[ ]];`

`<< MappaComplessa3.m`

(\* Lo usage di `PolarMap` e `CartesianMap` e' dichiarato tra `BeginPackage`

e `BeginPrivate`. \*) (\* `MakeLines` non ha usage (notare il colore blu) ;

e' ausiliare, nel contesto `Private` del pacchetto `MappaComplessa3.m` \*)

`? PolarMap`

`? CartesianMap`

`? MakeLines`

Symbol

`PolarMap[f, {r0:0, r1, (dr)}, {p0, p1, (dp)}]`

plots the image of the polar coordinate lines under the function `f`.

The default values of `dr` and `dp` are chosen so that the number of lines  
is equal to the value of the option `PlotPoints` of `Plot`.

▼

Symbol

`CartesianMap[f, {x0, x1, (dx)}, {y0, y1, (dy)}]`

plots the image of the cartesian coordinate lines under the function `f`.

The default values of `dx` and `dy` are chosen so that the number of lines  
is equal to the value of the option `PlotPoints` of `Plot`.

▼

`Missing[UnknownSymbol, MakeLines]`

```
(* Grazie a Get ,
$ContextPath e' stato aggiornato col Context MappaComplessa3` *)
$ContextPath
```

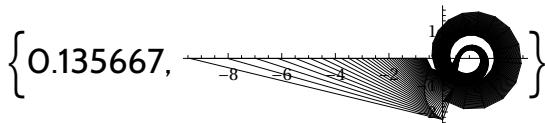
```
{DocumentationSearch`, ResourceLocator`, MappaComplessa3`, System`, Global`}
```

```
(* I nomi MappaComplessa3,
MakeLines sono possibili simboli nel Context Global` (notare il colore blu) *)
(* PolarMap e CartesianMap stanno nel Context MappaComplessa3` *)
Map[Context, {MappaComplessa3, MakeLines, PolarMap, CartesianMap}]
```

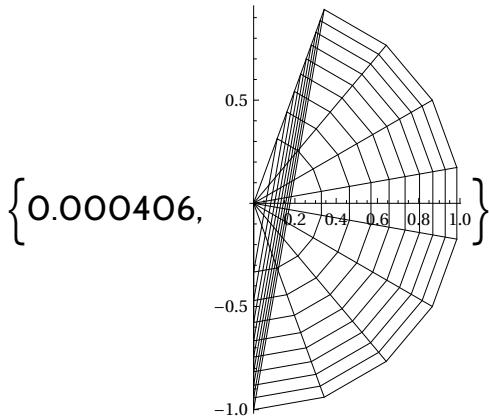
```
{Global`, Global`, MappaComplessa3`, MappaComplessa3`}
```

Esempi di uso di PolarMap e CartesianMap, nella versione corrente.

```
SetOptions[ Plot, PlotPoints → 50];
Timing[CartesianMap[Zeta, {0.1, 0.9}, {0, 20}]]
```



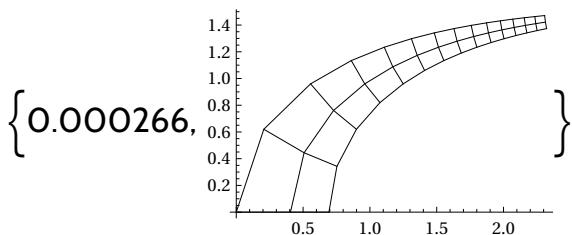
```
SetOptions[ Plot, PlotPoints → 10];
Timing[PolarMap[Sqrt, {1}, {-Pi + 0.0001, Pi}]]
```



Qui sotto, grafichiamo la immagine delle linee che uniscono coordinate in scala logaritmica (inversa della esponenziale).

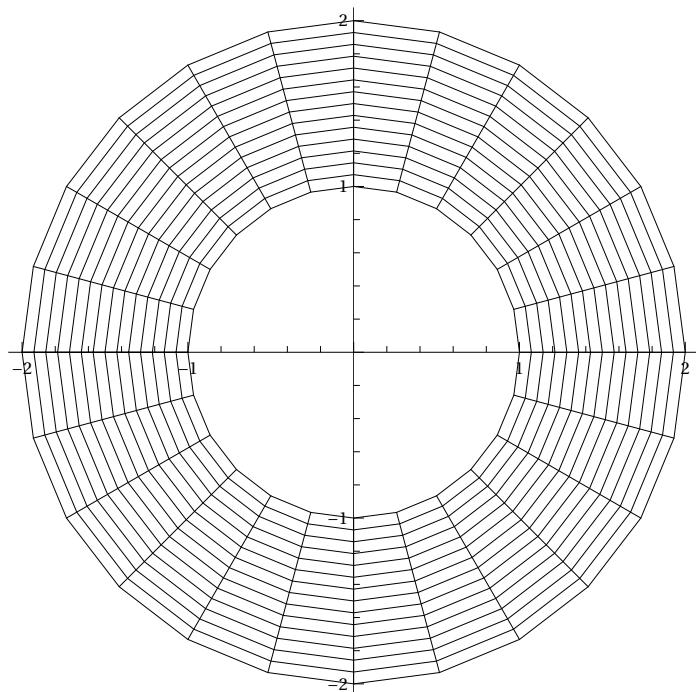
(\* Chiamata mista \*)

```
SetOptions[ Plot, PlotPoints → 15];
Timing[CartesianMap[Log, {1, 2, 1/2}, {0, 10}]]
```



(\* Chiamata mista \*)

```
SetOptions[ Plot, PlotPoints → 15];
PolarMap[Identity, {1, 2}, {-Pi, Pi, Pi/12}]
```



```
SetOptions[ Plot, PlotPoints → Automatic];
```

### Nota

I nomi per pattern come **dx** e **dy**, che sono argomenti di input della funzione che stiamo definendo (usati da SameQ nel codice di **MappaComplessa3.m**), non possono essere usati come variabili locali dentro al body della (regola

che definisce la funzione.

Pertanto, dichiariamo due variabili locali **ndx** e **ndy** e le inizializziamo coi valori di **dx** e **dy** rispettivamente.

### 1.5.1 Default o Options ??

I default sono convenienti, in quanto ci fanno risparmiare, ad esempio, sul dovere usare una scrittura ripetitiva del codice. Usare troppi default, però, può creare confusione.

In **PolarMap** (così come è scritta fin qui) esiste una possibile ambiguità, dovuta al fatto che (nel primo range, quello per il raggio **r**) è stato definito un default sia per il valore iniziale (**start**), sia per l'incremento.

Di conseguenza, c'è ambiguità sul significato della chiamata con due valori **{a, b}** :  
**(SI)** **a** è lo **start**, **b** è il valore finale (**final**), l'incremento è dato da default ? oppure

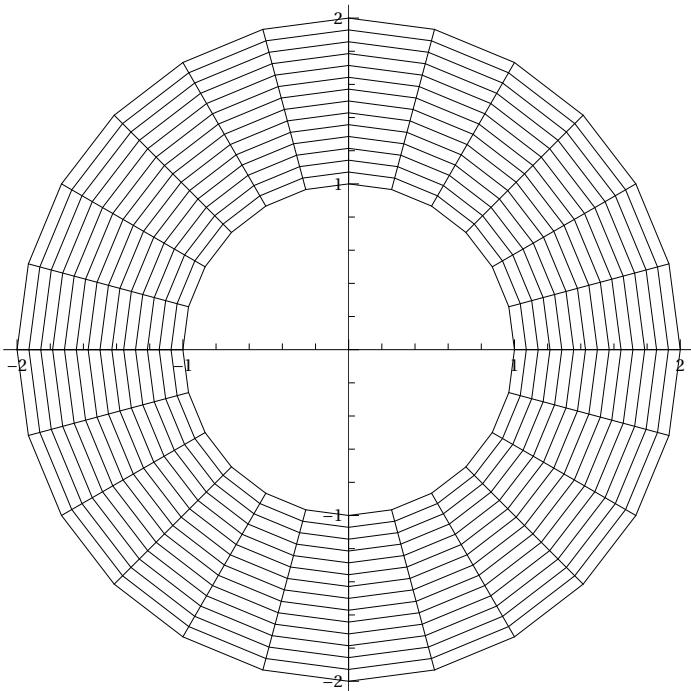
**(NO)** il valore di **start** è zero (come da default), **a** è il valore di **final**, **b** è l'incremento ?

→ In casi come questo, l'ambiguità si supera ricordando che la convenzione che si adotta è il matching da sinistra verso destra.

Nell'esempio qui sotto, quindi, il range **{a, b} = {1, 2}** sta ad indicare che il raggio **r** varia in **[a, b]**, in cui:

- **a=1** è lo **start**,
- **b=2** è il valore di **final**,
- mentre l'incremento è dato dal **default**, specificato tramite la Option **Plot-Points→15** (di **Plot**).

```
SetOptions[ Plot, PlotPoints → 15];
PolarMap[Identity, {1, 2}, {−Pi, Pi, Pi/12}]
```



✿ Gli argomenti (di una funzione) il cui significato dipende dalla loro posizione (nella lista degli argomenti) sono detti **argomenti posizionali**. Gli argomenti di **CartesianMap** e di **PolarMap** sono posizionali.

✿ Esistono anche i cosiddetti **argomenti nominali** (*named arguments*) : in *Mathematica*, essi sono le **Options**.

Gli argomenti nominali sono identificati dal loro nome (appunto), non dalla loro posizione (nella lista degli argomenti).

Essi pertanto possono stare in qualsiasi ordine .

Gli argomenti nominali (Options) sono preferibili quando una funzione ha molte caratteristiche settabili da utente (e, allo stesso tempo, è necessario dare, a tali caratteristiche, un default che vada bene nella maggiore parte dei casi e delle applicazioni).

Un esempio è costituito dalle funzioni di Grafica, nelle quali ci sono molte caratteristiche con valori di default che vanno benissimo per la maggiore parte delle situazioni e che , allo stesso tempo, è giusto poter variare occasionalmente.

(\* numero di Options di Plot, nella versione corrente di Mathematica \*)

Length[Options[Plot]]

(\* Le prime 7 Options di Plot coi relativi default \*)

Options[Plot][[1 ;; 7]] // TableForm

64

AlignmentPoint → Center

AspectRatio →  $\frac{1}{\text{GoldenRatio}}$

Axes → True

AxesLabel → None

AxesOrigin → Automatic

AxesStyle → {}

Background → None

**Nota.** In generale ,

- i valori di default sono gli argomenti “posizionali” (di chiamata) di una funzione ,

- mentre le Options sono gli argomenti “nominali” (di chiamata) di una funzione .

Per evitare ambiguità, è meglio mettere tutti i default (posizionali) dopo gli argomenti (di chiamata) obbligatori, se possibile.

(\* Qui, la funzione g e' definita con 4

argomenti: 2 obbligatori e 2 con default posizionali \*)

(\* La chiamata di g su due argomenti attiva i valori di default \*)

Clear[g];

g[x\_, y\_:1, z\_, t\_:2] := {x, y, z, t};

{g[], g[a], g[a, b], g[a, b, c], g[a, b, c, d]}

{g[], g[a], {a, 1, b, 2}, {a, b, c, 2}, {a, b, c, d}}

```

3 (* :Title: MappaComplessa3 *)
4 (* :Context: MappaComplessa3` *)
5 (* :Author: GS *)
6 (* :Summary: versione modificata di MappaComplessa2MakeLines *)
7 (* :Copyright: GS 2023 *)
8 (* :Package Version: 2.2 *)
9 (* :Mathematica Version: 13 *)
10 (* :History: last modified 31/3/2023 *)
11 (* :Sources: bbllio *)
12 (* :Limitations: educational purposes *)
13 (* :Discussion: *)
14 (* :Requirements: *)
15 (* :Warning: DOCUMENTATE TUTTO il codice *)
16
17 BeginPackage["MappaComplessa3`"]
18
19 CartesianMap::usage = "CartesianMap[f, {x0, x1, (dx)}, {y0, y1, (dy)}]
20 plots the image of the cartesian coordinate lines under the function f.
21 The default values of dx and dy are chosen so that the number of lines
22 is equal to the value of the option PlotPoints of Plot."
23
24 PolarMap::usage = "PolarMap[f, {r0:0, r1, (dr)}, {p0, p1, (dp)}]
25 plots the image of the polar coordinate lines under the function f.
26 The default values of dr and dp are chosen so that the number of lines
27 is equal to the value of the option PlotPoints of Plot."
28
29
30 Begin["`Private`"]

```

```

33 CartesianMap[ func_, {x0_, x1_, dx_:Automatic}, {y0_, y1_, dy_:Automatic}] :=
34   Module[ {x,y,coords,plotpoints, ndx=dx, ndy=dy},
35
36     plotpoints= PlotPoints/.Options[Plot];
37
38   If[ dx==Automatic, ndx=(x1-x0)/(plotpoints-1) ];
39   If[ dy==Automatic, ndy=(y1-y0)/(plotpoints-1) ];
40
41   coords = Table[ N[ func[ x+I y]], {x,x0,x1,ndx},{y,y0,y1,ndy}];
42
43   Show[MakeLines[coords], AspectRatio→Automatic, Axes→Automatic]
44 ]

```

---

```

47 PolarMap[ func_, {r0_:0, r1_, dr_:Automatic}, {p0_, p1_, dp_:Automatic}] :=
48   Module[ {r,p,coords,plotpoints, ndr=dr, ndp=dp},
49
50     plotpoints= PlotPoints/.Options[Plot];
51
52   If[ dr==Automatic, ndr=(r1-r0)/(plotpoints-1) ];
53   If[ dp==Automatic, ndp=(p1-p0)/(plotpoints-1) ];
54
55
56   coords = Table[ N[ func[ r Exp[ I p] ] ], {r,r0,r1,ndr},{p,p0,p1,ndp}];
57
58   Show[MakeLines[coords], AspectRatio→Automatic, Axes→Automatic]
59 ]

```

```
62 (* auxiliary function *)
63 MakeLines[points_] :=
64     Module[ {coords, lines},
65
66         coords = Map[ {Re[#], Im[#]}& , points, {2} ] ;
67
68         lines = Map[ Line, Join[ coords, Transpose[coords] ] ] ;
69
70         Graphics[ lines ]
71     ]
72
73 End[ ]
74
75 EndPackage[ ]
```

## 1.6 Passare Options ad un'altra funzione

Nelle funzioni **CartesianMap** e **PolarMap** viene usata la Built-in **Show**.

Show mostra più grafici combinati assieme in una sola area grafica, e ha varie Options specificabili.  
Show accetta qualsiasi Option applicabile ad una direttiva di grafica.

The screenshot shows the Mathematica Documentation Center with the search term "?Show" entered. The results for the **Show** function are displayed. The first result is a general description: "Show[*graphics*, *options*] shows graphics with the specified options added." Below it is another description: "Show[ $g_1, g_2, \dots$ ] shows several graphics combined." A blue box highlights the second description. At the bottom of the highlighted box, there is a note in parentheses: "(\* statement nel pacchetto MappaComplessa e varianti \*)". Below this note, the **Show** function is shown with its arguments: **Show[MakeLines[coords], AspectRatio→Automatic, Axes→Automatic]**.

Nel nostro caso, le Options di Show già attivate sono:

**AspectRatio→Automatic**,

**Axes→Automatic**,

ma vogliamo poterne attivare altre, ed eventualmente anche sovrascrivere **AspectRatio** e **Axes**.

❖ Vogliamo poter specificare le Options (usate in Show) nella chiamata a **CartesianMap** e **PolarMap**.

❖ Il pattern, che dovrà combaciare con queste Options, deve permettere l'uso di un numero flessibile di tali opzioni (incluso il caso di nessuna opzione attivata).

❖ Questo si ottiene con un pattern (col triplo blank) della forma **opts\_\_\_**.

Il valore di **opts** non viene usato da **CartesianMap** e neppure da **PolarMap**:  
esso viene semplicemente passato a **Show**.

Modifichiamo il file **MappaComplessa3.m**, e salviamolo in **MappaComplessa4.m**, in modo che contenga il seguente codice:

```
CartesianMap[func_, {x0_, x1_, dx_: Automatic}, {y0_, y1_, dy_: Automatic}, opts___]:=  
Module[{x, y, coords, plotpoints, ndx = dx, ndy = dy},  
  
plotpoints = PlotPoints /. Options[Plot];  
  
If[dx === Automatic, ndx = (x1 - x0)/(plotpoints - 1)];  
If[dy === Automatic, ndy = (y1 - y0)/(plotpoints - 1)];  
  
coords = Table[N[func[x + I y]], {x, x0, x1, ndx}, {y, y0, y1, ndy}];  
  
Show[MakeLines[coords], opts, AspectRatio → Automatic, Axes → Automatic]  
]
```

Per **PolarMap** faremo una modifica analoga, mantenendo inoltre il valore zero come default per **r0** ossia

r0\_0.

Notiamo anche la posizione di `opts` nella lista degli argomenti di `Show[]`: aver messo `opts` prima delle regole `AspectRatio→Automatic`, `Axes→Automatic`, permette di sovrascrivere anche queste due opzioni (tramite chiamata a `CartesianMap` e a `PolarMap`).

```
(* Quit del Kernel *)
```

```
Quit[ ]
```

```
SetDirectory[NotebookDirectory[]];
<< MappaComplessa4.m
```

```
(* Lo usage di PolarMap e CartesianMap e' dichiarato tra BeginPackage e BeginPrivate. *)
```

```
(* MakeLines non ha usage (notare il colore blu); e' ausiliaria, nel contesto Private del pacchetto MappaComplessa4.m *)
```

```
?PolarMap
```

```
?CartesianMap
```

```
?MakeLines
```

```
(* Grazie a Get, $ContextPath e' stato aggiornato col Context MappaComplessa4` *)
```

```
$ContextPath
```

```
{DocumentationSearch`, ResourceLocator`, MappaComplessa4`, System`, Global`}
```

```
(* I nomi MappaComplessa4 e MakeLines sono possibili simboli nel Context Global` (notare il colore blu) *)
```

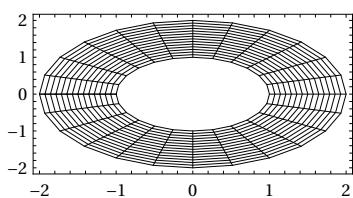
```
(* PolarMap e CartesianMap stanno nel Context MappaComplessa4` *)
```

```
Map[Context, {MappaComplessa4, MakeLines, PolarMap, CartesianMap}]
```

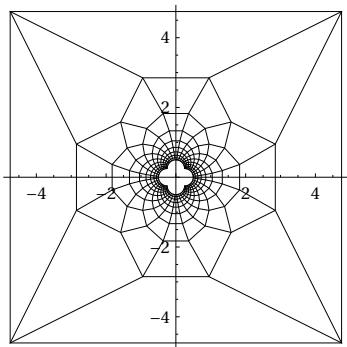
```
{Global`, Global`, MappaComplessa4`, MappaComplessa4`}
```

Esempi di uso di `PolarMap` e `CartesianMap`, nella versione corrente.

```
SetOptions[Plot, PlotPoints → 15];
PolarMap[ Identity, {1, 2}, {-Pi, Pi, Pi/12},
  AspectRatio → 0.5,
  Axes → None,
  Frame → True,
  ImageSize → Small]
```



```
SetOptions[ Plot, PlotPoints → 15];
CartesianMap[ 1/Conjugate[#] &, {−2, 2, 4/19}, {−2, 2, 4/19},
  PlotRange → All,
  ImageSize → Small]
```



```
SetOptions[ Plot, PlotPoints → Automatic];
```

```
3 (* :Title: MappaComplessa4 *)
4 (* :Context: MappaComplessa4` *)
5 (* :Author: GS *)
6 (* :Summary: a version of the ComplexMap package *)
7 (* :Copyright: GS 2023 *)
8 (* :Package Version: 2 *)
9 (* :Mathematica Version: 13 *)
10 (* :History: last modified 31/3/2023 *)
11 (* :Sources: bbllio *)
12 (* :Limitations: educational purposes *)
13 (* :Discussion: Passare Options a routine ausiliarie *)
14 (* :Requirements: *)
15 (* :Warning: DOCUMENTATE TUTTO il codice *)
```

```
18 BeginPackage["MappaComplessa4`"]
19
20 CartesianMap::usage = "CartesianMap[f, {x0, x1, (dx)}, {y0, y1, (dy)}]
21     plots the image of the cartesian coordinate lines under the function f.
22     The default values of dx and dy are chosen so that the number of lines
23     is equal to the value of the option PlotPoints of Plot[ ]."
24
25 PolarMap::usage = "PolarMap[f, {r0:0, r1, (dr)}, {p0, p1, (dp)}]
26     plots the image of the polar coordinate lines under the function f.
27     The default values of dr and dp are chosen so that the number of lines
28     is equal to the value of the option PlotPoints of Plot[ ]."
31 Begin["`Private`"]

```

```

34 CartesianMap[ func_, {x0_, x1_, dx_:Automatic}, {y0_, y1_, dy_:Automatic}, opts__] :=
35   Module[ {x,y,coords,plotpoints, ndx=dx, ndy=dy},
36
37     plotpoints= PlotPoints/.Options[Plot];
38
39   If[ dx==Automatic, ndx=(x1-x0)/(plotpoints-1) ];
40   If[ dy==Automatic, ndy=(y1-y0)/(plotpoints-1) ];
41
42   coords = Table[ N[ func[ x+I y]], {x,x0,x1,ndx},{y,y0,y1,ndy}];
43
44 Show[MakeLines[coords], opts, AspectRatio→Automatic, Axes→Automatic]
45 ]

```

---

```

48 PolarMap[ func_, {r0_:0, r1_, dr_:Automatic}, {p0_, p1_, dp_:Automatic} ,opts__] :=
49   Module[ {r,p,coords,plotpoints, ndr=dr, ndp=dp},
50
51     plotpoints= PlotPoints/.Options[Plot];
52
53   If[ dr==Automatic, ndr=(r1-r0)/(plotpoints-1) ];
54   If[ dp==Automatic, ndp=(p1-p0)/(plotpoints-1) ];
55
56   coords = Table[ N[ func[ r Exp[ I p ] ] ], {r,r0,r1,ndr},{p,p0,p1,ndp}];
57
58 Show[MakeLines[coords], opts, AspectRatio→Automatic, Axes→Automatic]
59 ]

```

---

```

61 (* auxiliary function *)
62 MakeLines[points_] :=
63   Module[ {coords, lines},
64
65     coords = Map[ {Re[##], Im[##]}& , points, {2} ];
66
67     lines = Map[ Line, Join[ coords, Transpose[coords] ] ] ;
68
69     Graphics[ lines ]
70   ]

```

73 End[ ]

76 EndPackage[ ]

## Module e Block

Uno dei modi per essere sicuri che parti differenti di un programma non interferiscano è quello di dare alle rispettive variabili una determinata “visibilità” (scope).

Nei linguaggi dotati del concetto di blocco (come C, C++, Pascal, Java e molti altri), una variabile visibile all'interno di un blocco è in generale visibile anche all'interno di eventuali blocchi annidati.

Le regole fondamentali di visibilità sono in genere modificate dalla regola speciale dello “shadowing”, secondo cui una variabile locale “nasconde” una eventuale variabile omonima definita nello scope superiore.

In altre parole, se in un programma è definita una variabile globale e in un determinato sottoprogramma viene definita una variabile locale omonima, il sottoprogramma in questione perde la visibilità della variabile globale, nascosta da quella locale.

Module corrisponde a **scoping statico o lessicale** (localizza i nomi di variabile): le variabili sono trattate come locali ad una determinata sezione del codice (in un programma).

Block corrisponde a **scoping dinamico** (localizza i valori delle variabili): le variabili sono trattate come locali ad una determinata parte della history di esecuzione (di un programma).

### ■ Esempio

#### ■ Block

```
(* Considero z e x variabili globali *)
(* Non assegno valori alla z nel contesto globale *)
(* Assegno un valore ad x nel contesto globale : x=1 *)
Clear[z, x]; z; x = 1; --> faccio clear per sicurezza
Print["Prima di Block: x = ", x, ", z = ", z];

Block[{x}, (* x è dichiarata locale (x verde)
    => il simbolo x locale ha lo stesso nome del simbolo x globale,
        ma i loro contenuti sono diversi
        (il contenuto di x globale è salvato a parte)
        Non assegno valori a x dentro Block. *)
z = x; (* z non è dichiarata locale
    => z rimane nota a livello globale (z nera, dopo questa Set)
    => la sua assegnazione z=x è nota globalmente,
        anche se avviene dentro Block *)
Print["Dentro Block: x = ", x, ", z = ", z];
];

Print["Finita Block: x = ", x, ", z = ", z];
(* x torna simbolo globale
=> il contenuto di x torna quello che era prima di Block (x=1) *)

(* z è rimasta nota a livello globale anche dentro Block
=> z non è più non-assegnata (come era prima di Block):
    z ha il contenuto (z=x) noto globalmente (anche se assegnato dentro Block) *)

x = 2; (* Se ri-assegno x , ri-assegno anche z *)
Print["Ri-assegnata x: x = ", x, ", z = ", z];

```

Prima di Block: x = 1, z = z

Dentro Block: x = x, z = x

Finita Block: x = 1, z = 1

Ri-assegnata x: x = 2, z = 2

## ■ Module

```

(* Considero z e x variabili globali *)
(* Non assegno valori a z nel contesto globale *)
(* Assegno un valore a x nel contesto globale : x=1 *)
Clear[z, x];   x = 1;
Print["Prima di Module: x = ", x, ",          z = ", z];

Module[ {x}, (* x è dichiarata locale (x verde)
    ⇒ viene creato x$num locale, diverso da x globale.
    Non assegno valori a x$num dentro Module *)
z = x;  (* z non è dichiarata locale
    ⇒ z rimane nota a livello globale (z nera, dopo questa Set)
    ⇒ la sua assegnazione z=x$num è nota globalmente,
    anche se avviene dentro Module *)
Print["Dentro Module:  x = ", x, ",  z = ", z];
];

Print["Finita Module:  x = ", x, ",          z = ", z];
(* x è (sempre rimasto) simbolo globale, col suo contenuto *)

(* z è rimasta nota a livello globale anche dentro Module
⇒ z non è più non-assegnata (come era prima di Module):
z ha il contenuto (z=x$num) noto globalmente
(anche se assegnato dentro Module) *)

x = 2;  (* Se ri-assegno x , non ri-assegno z *)
Print["Ri-assegnata x:  x = ", x, ",          z = ", z];

```

Prima di Module: x = 1, z = z

Dentro Module: x = x\$4682, z = x\$4682

Finita Module: x = 1, z = x\$4682

Ri-assegnata x: x = 2, z = x\$4682

Riporto qui gli output ottenuti con Block, per comodita' di paragone

Prima di Block: x = 1, z = z

Dentro Block: x = x, z = x

Finita Block: x = 1, z = 1

Ri-assegnata x: x = 2, z = 2

## ■ Come funziona Module

Ogni volta che Module è usata, viene creato un nuovo simbolo per rappresentare ciascuna delle sue variabili locali.

A tale nuovo simbolo viene dato un nome univoco, che non può entrare in conflitto con nessun altro nome.

Tale nome univoco è formato dal nome della variabile locale stessa, seguito da \$num (in cui num è un numero seriale, che viene progressivamente incrementato).

```
Clear[t, tglobal];
t = tglobal;
Print["t prima di Module: ", t];

Module[{t},
  Print["t dentro Module: ", t];
  t];

Print["t dopo Module: ", t];
```

```
t prima di Module: tglobal
t dentro Module: t$4661
t dopo Module: tglobal
```

Posso usare il simbolo t, creato e restituito dalla Module, anche fuori da essa:

```
Clear[t];
outm = Module[{t}, t];
1 + outm^2

1 + t$4690*
```

Esempio 2 (non nec)

Esempio 3 (non nec)

## ■ Come funziona Block

Block crea una parte di codice (un environment) in cui è possibile cambiare temporaneamente il **valore** delle variabili di interesse.

Espressioni, che vengono valutate durante l'esecuzione di una Block, usano i valori definiti correntemente (localmente nella Block).

Questo è vero sia nel caso in cui tali espressioni appaiano direttamente (come parti di un body di Block), sia nel caso in cui tali espressioni vengano prodotte dalla valutazione stessa.

```

Clear[t, tglobal];
t = tglobal;
Print["t prima di Block: ", t];

Block[{t},
  Print["t dentro Block: ", t];
  t];
(* Anche se t è stata assegnato globalmente al valore tglobal,
qui ho dichiarato t locale a Block e non gli ho dato alcun valore
==> quindi Print[t] restituisce il simbolo t
che è il valore localmente dato a t dentro Block .

==> Block restituisce il simbolo t ,
il quale (appena Block termina) viene valutato e riconosciuto
come simbolo globale , cui è assegnato il valore tglobal *)
Print["t dopo Block: ", t];

```

```

t prima di Block: tglobal
t dentro Block: t
t dopo Block: tglobal
(* Riporto gli output ottenuti con Module, per comodita' di paragone *)

```

```

t prima di Module: tglobal
t dentro Module: t$4661
t dopo Module: tglobal

```

Posso usare il simbolo t (dichiarato locale alla Block), anche fuori da essa:

```

Clear[t];
outb = Block[{t}, t];
1 + outb^2
1 + t^2

```

Esempio 2 (non nec)

Esempio 3 (non nec)

- Quando è utile Block?

- In costrutti iterativi (e.g. Do, Sum, Table)
  - Block, per lavorare in Precision pre - fissata

# Introduction to Dynamic

This tutorial describes the principles behind **Dynamic** and **DynamicModule** and related functions, and it goes into detail about how they interact with each other and with the other functionalities.

These functions are the foundation of the higher –level function **Manipulate**, that provides a simple way for creating interactive examples, programs, and Demonstrations, in a convenient (though relatively rigid) structure.

If that structure solves your problem, use **Manipulate**.

Read this tutorial, if you want to build a wider range of structures, including your own (variably complex) user interfaces.

**Note1** : in this tutorial (as in all material in VIRTUALE),  
you are expected to evaluate all the input lines as you reach them, and watch what happens;  
the accompanying text will not make sense without evaluating as you read.

**Note2** : for In/Out labelling : quit the kernel, then go to menu

Format/Option Inspector /

Selection : yourNotebook.nb + Cell Options / Cell Labels / ShowCellLabels → Automatic  
(for a single cell)

cell : select it and use menu Cell/Cell Tags / Show Cell Tags + Cell Tags from In/Out Names )

---

## The Fundamental Principle of Dynamic

An ordinary *Mathematica* session consists of a series of static inputs and outputs, which form a record of calculations, done in the (temporal) order in which they were entered.

```
In[15]:= (* Evaluate each of these inputs one after the other *)
x = 5; x2
```

```
Out[15]= 25
```

```
In[16]:= x = 7; x2
```

```
Out[16]= 49
```

The first output shows the value from when **x** was 5 (even though **x** is now 7). This is, of course, useful, if we want to see a history of what we have been doing.

However, we may want an output that is automatically updated, to always reflect its current value. This kind of output is provided by **Dynamic**.

```
In[17]:= (* Evaluate the following cell;
note that the result will be 49 because the current value of x is 7 *)
Dynamic[x^2]
Out[17]= 1
```

Generally, when we first evaluate an input that contains a variable wrapped in **Dynamic**, we will get the same result as without **Dynamic**.

But, if we subsequently change the value of the variable, the displayed output will change retroactively.

```
In[18]:= (* Evaluate the following cells one at a time,
and note the change in the value of
the Dynamic Output Cell above (possibly, Out[5]) *)
x = 9;
In[19]:= x = 15;
In[20]:= x = 10;
```

The very first two static outputs are still 25 and 49 respectively, but the single dynamic output now displays 100, the square of the last value of **x** (this last sentence will, of course, become incorrect as soon as the value of **x** is changed again).

There are no restrictions on the types of values that can go into a **Dynamic**.

Even though **x** was initially a number, **x** can become a formula, or a graphic, in subsequent evaluations.

This simple feature is the basis for a rich set of interactive capabilities.

```
In[21]:= Dynamic[x]
Out[21]= 1
In[22]:= (* Each time the value of x is changed,
the value of Dynamic[x] and Dynamic[x^2] above is updated automatically *)
x = Integrate[ $\frac{1}{1-y^3}$ , y];
In[23]:= x = Plot[Sin[x], {x, 0, 2 Pi}, ImageSize -> Tiny];
In[24]:= x = 0;
```

**Dynamic [expression]** is an object that displays as the (dynamically updated) current value of **expression**.

# Dynamic and Controls

**Dynamic** is often used with Controls such as Sliders and Checkboxes.

The full range of controls available in *Mathematica* is discussed in:

```
In[12]:= Hyperlink\["https://reference.wolfram.com/language/guide/ControlObjects.html"\]
Out[12]= https://reference.wolfram.com/language/guide/ControlObjects.html
```

Here Sliders are used to illustrate how things work.

The principles of using **Dynamic** with other Controls is basically the same.

The built-in **Slider** needs a first argument to specify the position of the thumb (cursore); (a second argument is optional and it specifies range + step size {start, stop, step} : default range is 0 to 1; default step size is 1).

```
In[25]:= (* This is a slider centered in the middle of [0, 1] *)
sliderStatic = Slider[0.5]
Out[25]=
```



Drag **sliderStatic** around.

The thumb moves, but nothing else happens, since **sliderStatic** is not connected to anything.

The following statement associates the initial position of the slider with the current value of **x** (we thus call this **x-slider**):

```
In[29]:= sliderDyn = Slider[ Dynamic[x] ]
Out[29]=
```



```
In[31]:= (* This re-creates a Dynamic output of x *)
Dynamic[x]
Out[31]=
```

1

Drag **sliderDyn** around.

As the slider moves, the value of **x** changes and the Dynamic output updates in real time.

The slider **sliderDyn** also responds to changes in the value of **x**.

To see this, evaluate the following statement:

In[34]:= **x = 0.8;**

You should see , simultaneously :

- the slider **sliderDyn** jump ;
  - the Dynamic output of **x** change.
- 

This creates another x-slider :

In[35]:= **sliderDyn2 = Slider[ Dynamic[x] ]**

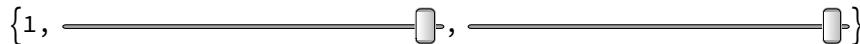
Out[35]=



If we move one of the two sliders, **sliderDyn** or **sliderDyn2**, the other one moves in lock-sync . Both are connected (dynamically and bi-directionally) to the current value of **x** .

In[41]:= **{Dynamic[x], sliderDyn, sliderDyn2}**

Out[41]=



## Dynamic and Other Functions

**Dynamic** and Control constructs, such as **Slider**, are like any other function in *Mathematica*.

They can occur anywhere, e.g. in Tables or inside a mathematical expression.

Wherever these functions occur, they carry with them the behavior of Dynamically displaying (or changing in real time) the current value of the variable (or expression) they are linked to.

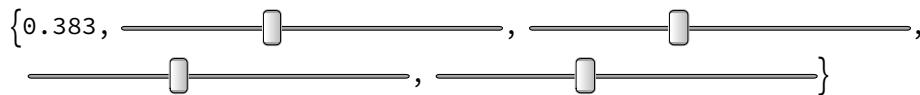
**Dynamic** is a building block; combined with other *Mathematica* functionalities, it becomes a flexible tool for creating interactive displays.

---

This makes a Table of y-sliders, which are updated in sync :

In[36]:= **{ Dynamic[y], Table[ Slider[Dynamic[y]], {4} ] } // Flatten**

Out[36]=



As seen above, we can combine a Slider with a display of its current value, in a single output :

In[37]:= **{Slider[Dynamic[q]], Dynamic[q]}**

Out[37]=



**Dynamic** can display any function of a variable **t** :

```
In[38]:= {Slider[Dynamic[t]], Dynamic[ Plot[Sin[10 θ t], {θ, 0, 2 Pi}, ImageSize → Tiny] ] }
```

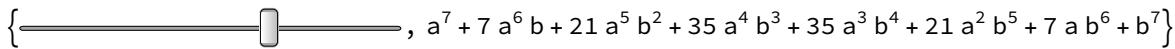
Out[38]=



By using a slider that takes **Integer** values, we can dynamically update algebraic expressions .

```
In[39]:= {Slider[ Dynamic[p], {1, 10, 1} ], Dynamic[ Expand[(a+b)^p] ] }
```

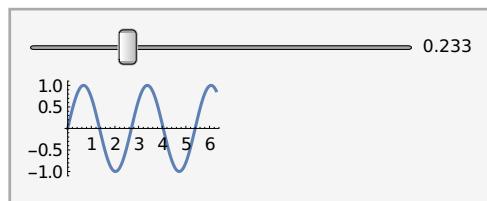
Out[39]=



We can use Dynamic expressions with formatting constructs, like **Panel**, **Row**, **Column**, **Grid**.

```
(* Creiamo un pannello *)
Panel[
(* Mettiamo in Colonna uno Slider ed una Plot *)
Column[
{
(* Mettiamo in una Riga lo Slider ed il suo valore corrente *)
Row[ {Slider[Dynamic[k]], " ", Dynamic[k]} ],
Dynamic[ Plot[ Sin[10 θ k], {θ, 0, 2 Pi}, ImageSize → Tiny] ]
}
]
]
```

Out[72]=



The last example resembles the output of **Manipulate**.

This is no coincidence: **Manipulate** , in fact, produces a combination of **Dynamic** , Controls and formatting constructs, similar to what we can obtain using these lower-level functions.

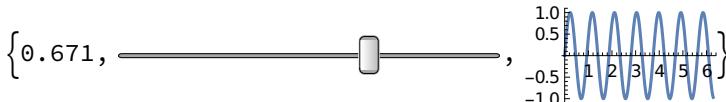
## Localizing Variables in Dynamic Output

Here is another **z**-slider connected to a function of **z**

( Plot of  $\text{Sin}[10 y z]$ , in which  $y$  is a parameter of Plot ):

```
In[74]:= egSync1 = {Dynamic[z], Slider[Dynamic[z]],  
Dynamic[Plot[Sin[10 y z], {y, 0, 2 Pi}, ImageSize -> Tiny]]}
```

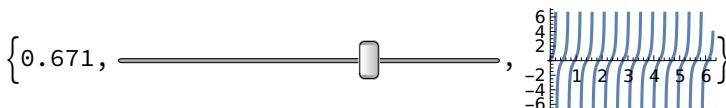
Out[74]=



This is a **z**-slider connected to another function of **z** ( Plot of  $\text{Tan}[10 y z]$  ):

```
In[75]:= egSync2 = {Dynamic[z], Slider[Dynamic[z]],  
Dynamic[Plot[Tan[10 y z], {y, 0, 2 Pi}, ImageSize -> Tiny]]}
```

Out[75]=



The two **z**-sliders are communicating with each other  
(move the slider in one example, and the other example moves too).

This is because we are using the global variable **z** in both examples.

Although this can be useful in some situations, most of the time we would probably be happier if these two sliders could be moved independently.

The solution is the function **DynamicModule**.

**DynamicModule**[ $\{x, y, \dots\}$ , **expression**] is an object which maintains the same local instance of symbols **x**, **y**, etcetera, in the course of all evaluations of **Dynamic** objects in **expression**.

**DynamicModule**[ $\{x = x_0, y = y_0, \dots\}$ , **expression**] specifies initial values for **x**, **y**, etcetera.

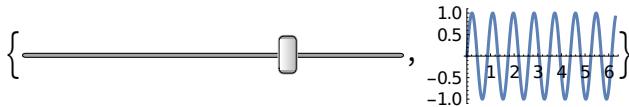
**DynamicModule** has arguments identical to **Module** and (like **Module**) it is used to localize (and initialize) variables.

But there are important differences in how they operate.

Here are the same two examples of **z**-slider as before, but with private values of **z**:

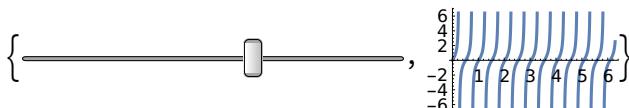
```
In[76]:= egSync1 = {Slider[Dynamic[z]], Dynamic[Plot[Sin[10 y z], {y, 0, 2 Pi}, ImageSize -> Tiny]]};
egDyn1 = DynamicModule[ {z = .5}, egSync1 ]
```

Out[77]=



```
In[78]:= egSync2 = {Slider[Dynamic[z]], Dynamic[Plot[Tan[10 y z], {y, 0, 2 Pi}, ImageSize -> Tiny]]};
egDyn2 = DynamicModule[ {z = .5}, egSync2 ]
```

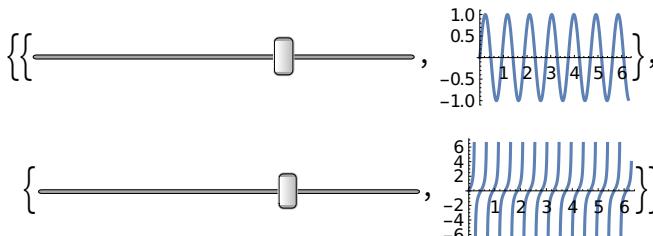
Out[79]=



The two examples **egDyn1** and **egDyn2** work independently of each other  
(unlike the synchronized examples **egSync1** and **egSync2**).

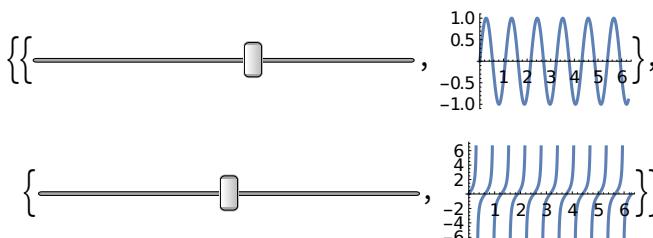
In[80]:= {egSync1, egSync2}

Out[80]=



In[81]:= {egDyn1, egDyn2}

Out[81]=



As seen, multiple instances of **DynamicModule** can be placed in a single output.  
They maintain separate values of the variables associated with their respective areas.

Basically, **Module** is useful for evaluation purposes (and its scope refers to the Kernel history session),

while **DynamicModule** is useful for dynamic purposes and the construction of an interactive interface

(and its scope refers mostly to the FrontEnd history session).

**DynamicModule** is the Dynamic analog of the **Module** function.

**DynamicModule** allows to localize (and initialize) variables in **Dynamic** objects and in **Dynamic** expression.

NOTE . Structure of the Mathematica System:

Kernel = part that actually performs computations

FrontEnd = part that handles interaction with the user

The Front End is the entire GUI (graphical user interface) to Mathematica .

It is the implementation of the Notebook Interface .

(The alternative is a text - based (command line/ text from the keyboard) interface) .

---

### NOTE.

Using **Module** (in place of **DynamicModule**) would appear to work, at first.

But it is **not** a good idea, for reasons discussed in:

```
In[29]:= Hyperlink[
  "https://reference.wolfram.com/language/tutorial/AdvancedDynamicFunctionality.html"]
Out[29]= https://reference.wolfram.com/language/tutorial/AdvancedDynamicFunctionality.html
```

---

### Module versus DynamicModule.

#### Example: Module versus DynamicModule.

In the following statement, with **Module**, the red **x** indicates that something may be wrong (even though the Slider seems to work).

```
In[91]:= Module[{x}, Slider[ Dynamic[x] , Appearance -> "Labeled" ]]
Out[91]=
```



The statement with **DynamicModule** signals no problem.

```
In[92]:= DynamicModule[{x}, Slider[ Dynamic[x] , Appearance -> "Labeled" ]]
Out[92]=
```



◆ Copy and paste the output cell (i.e. the cell that displays the Slider) obtained with **Module**. Move the pasted Slider .

```
Out[91]=
```



The original slider (obtained with **Module**) moves in sync with the pasted slider.

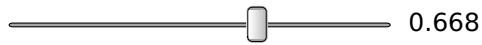
This happens because (even though **x** is a variable scoped with **Module**, and stored in the

Kernel),

the **x** of the original and pasted sliders (obtained with **Module**) are stored in the same way in the Kernel.

◆ Now, copy and paste the output cell (i.e. the cell that displays the Slider) obtained with **DynamicModule**. Move the pasted Slider.

Out[82]=



The original and pasted sliders (obtained with **DynamicModule**) moves independently from each other.

This happens because **x** of the original slider is localized independently (due to **DynamicModule**) from the **x** of the pasted slider.

◆ Furthermore, with **Module**, **x** is stored in the Kernel, therefore if we **Quit** the Kernel, then we loose the result of the Evaluation of the **Module** statement.

As a consequence, the Slider (original or pasted) obtained with **Module** does not work any longer.

In[3]:= **Quit**

Out[40]=



On the contrary, with **DynamicModule**, **x** is stored in the FrontEnd, therefore the Slider (original or pasted) obtained with **DynamicModule** still work, even after quitting the Kernel.

This means, as an example, that we can take our notebook, close, send it to someone else, who will open it and will be able to use our slider (or, in general, the interactive interface that we have built).

One of the advantages of **DynamicModule** is that it saves **state**.

This means (as said already) that we can end our *Mathematica* session, close the notebook, restart *Mathematica*, re-open the notebook, and any output created with a **DynamicModule** will be in the same state as when we closed the notebook.

This is because the output of a **DynamicModule** includes an expression (embedded in the output) that is initialized when it is displayed again; such an expression includes values of local variables created with **DynamicModule**.

## 1.7 Controllare le singolarità

Fin qui abbiamo considerato esempi in cui le funzioni da plottare non hanno singolarità nei punti di griglia (usati dalle built-in di grafica).

Le singolarità sono tuttavia una caratteristica delle funzioni a valori complessi.

Un esempio è dato dalla funzione inversa del cerchio unitario;

per graficare tale funzione bisogna stare attenti ad evitare un plot-range che contenga l'origine

(dato che, nell'origine, la funzione non è definita e, di conseguenza, Plot genererebbe una divisione per zero).

Anche valori molto grandi possono creare problemi (numerici).

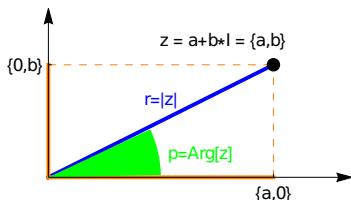
### 1.7.1 Valori di funzione enormi

□ Sostituiamo ogni numero  $z$  (complesso, nel nostro caso), il cui valore assoluto  $\text{Abs}[z]$  sia maggiore di una determinata soglia huge con un altro numero avente :

- argomento (angolo di fase  $p$ ) uguale all'argomento di  $z$  ;
- valore assoluto (modulo  $r$ ) uguale a **huge**, quindi minore di  $\text{Abs}[z]$ .

In questo modo, i due numeri hanno entrambi la stessa direzione nel piano complesso.

□ NOTA. Rappresentazioni di un numero complesso  $z = a+b\text{i}$  ovvero  $z = r * \text{Exp}[p\text{i}]$



Modifichiamo la funzione **MakeLines[]**, aggiungendo una soglia huge, memorizzata in una variabile locale.

La quantità  **$z/Abs[z]$**  restituisce un numero complesso normalizzato (i.e. con magnitudine 1) e avente lo stesso argomento di  $z$ .

Moltiplicandolo per il valore **huge**, otteniamo il numero voluto (che ha la stessa direzione di  $z$ , ma magnitudine minore).

Ripetiamo questo per ogni numero maggiore di **huge** in valore assoluto, usando la Condition  $\text{Abs}[z] > \text{huge}$ .

□ Modifichiamo anche i numeri troppo piccoli (che hanno valore assoluto vicino a zero), sostituendoli con zero, usando come Condition  $\text{Abs}[z] < 1/\text{huge}$ .

```
huge = 10^6;
MakeLines[points_] :=
Module[{coords, lines, newpoints},
newpoints = points /. z_?NumberQ :> huge * z/Abs[z] /; Abs[z] > huge;
newpoints = points /. z_?NumberQ :> 0.0 /; Abs[z] < 1/huge;
coords = Map[{Re[##], Im[##]} &, newpoints, {2}];
lines = Map[Line, Join[coords, Transpose[coords]]];
Graphics[lines];
]
```

- **NOTA.** Alcune forme di infinito hanno una direzione ad esse associata.

La più comune è **Infinity** che sta per  $+\infty$

Internamente, esse sono rappresentate con **DirectedInfinity[z]** in cui **z** è (di solito) un numero complesso (e il suo argomento è la direzione di tale infinito).

```
Map[FullForm, {Infinity, -Infinity, ComplexInfinity, ComplexInfinity === 1/0}]
```

(\* ComplexInfinity ha parte Reale e parte Immaginaria indeterminate \*)

```
{Re[Infinity], Im[Infinity], Re[ComplexInfinity], Im[ComplexInfinity]}
```

... Power: Infinite expression  $\frac{1}{0}$  encountered.

```
{DirectedInfinity[1], DirectedInfinity[-1], DirectedInfinity[], True}
```

```
{\infty, 0, Indeterminate, Indeterminate}
```

?DirectedInfinity

?ComplexInfinity

?Infinity

Per graficare linee verso tali punti, li sostituiamo semplicemente con un numero finito avente la stessa direzione.

```
huge = 10^6;
MakeLines[points_] :=
Module[{coords, lines, newpoints},
newpoints = points /.
{z_?NumberQ :> huge * z/Abs[z] /; Abs[z] > huge,
z_?NumberQ :> 0.0 /; Abs[z] < 1/huge,
DirectedInfinity[z_] :> huge * z/Abs[z]};

```

```
coords = Map[{Re[\#], Im[\#]} &, newpoints, {2}];
```

```
lines = Map[Line, Join[coords, Transpose[coords]]];
```

```
Graphics[lines];
]
```

- Queste modifiche sono nel file [MappaComplessa5.m](#)

```
(*Quit del Kernel*)
```

```
Quit[];
```

```
SetDirectory[NotebookDirectory[]];
```

```
<< MappaComplessa5.m
```

```
(* Lo usage di PolarMap e CartesianMap e' dichiarato tra BeginPackage e BeginPrivate. *)
(* MakeLines non ha usage (notare il colore blu) ; e' ausiliaria, nel contesto Private del pacchetto MappaComplessa5.m *)
?PolarMap
?CartesianMap
?MakeLines
```

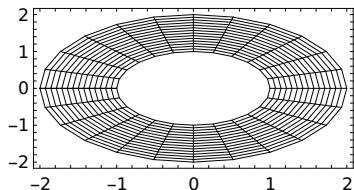
```
(*Grazie a Get,$ContextPath e' stato aggiornato col Context MappaComplessa5`*)
(* I nomi MappaComplessa5 e MakeLines sono possibili simboli nel Context Global` (notare il colore blu) *)
(* PolarMap e CartesianMap stanno nel Context MappaComplessa5` *)
{$ContextPath, Context[ComplexMap], Context[MakeLines], Context[PolarMap], Context[CartesianMap]}
```

```
{DocumentationSearch, ResourceLocator, MappaComplessa5`, System`, Global`},
Global`, Global`, MappaComplessa5`, MappaComplessa5`}
```

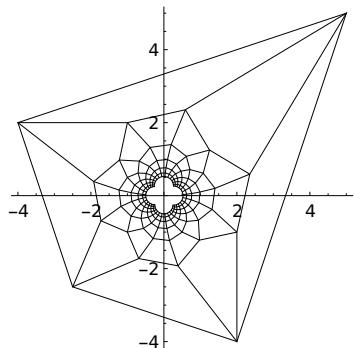
Esempi di uso di PolarMap e CartesianMap, nella versione corrente .

```
SetOptions[ Plot, PlotPoints → 15];
```

```
PolarMap[ Identity, {1, 2}, {-Pi, Pi, Pi/12},
AspectRatio → 0.5, Axes → None, Frame → True]
```



```
CartesianMap[ 1/Conjugate[#] &, {-2, 2, 0.3}, {-2, 2, 0.3},
PlotRange → All]
```



```
SetOptions[ Plot, PlotPoints → Automatic];
```

```
3 (* :Title: MappaComplessa5 *)
4 (* :Context: MappaComplessa5` *)
5 (* :Author: GS *)
6 (* :Summary: a version of the ComplexMap package *)
7 (* :Copyright: GS 2023 *)
8 (* :Package Version: 2 *)
9 (* :Mathematica Version: 13 *)
10 (* :History: last modified 31/3/2023 *)
11 (* :Sources: bbllio *)
12 (* :Limitations: educational purposes *)
13 (* :Discussion: Controllo su valori molto grandi o molto piccoli *)
14 (* :Requirements: *)s
15 (* :Warning: DOCUMENTATE TUTTO il codice *)
```

```
18 BeginPackage["MappaComplessa5`"]
19
20 CartesianMap::usage = "CartesianMap[f, {x0, x1, (dx)}, {y0, y1, (dy)}]
21     plots the image of the cartesian coordinate lines under the function f.
22     The default values of dx and dy are chosen so that the number of lines
23     is equal to the value of the option PlotPoints of Plot[ ]."
24
25 PolarMap::usage = "PolarMap[f, {r0:0, r1, (dr)}, {p0, p1, (dp)}]
26     plots the image of the polar coordinate lines under the function f.
27     The default values of dr and dp are chosen so that the number of lines
28     is equal to the value of the option PlotPoints of Plot[ ]."
29
30 Begin["`Private`"]

```

```

33 (* auxiliary function *)
34 huge = 10.0^6;
35 huge0=10.0^3;
36 MakeLines[ points_] :=
37   Module[ {coords, lines, newpoints},
38
39   newpoints = points /.
40   {z_?NumberQ → huge z/Abs[z] /; Abs[z] > huge,
41   z_?NumberQ → 0.0 /; Abs[z] < 1/huge,
42   DirectedInfinity[z_] → huge z/Abs[z],
43   DirectedInfinity[] → huge0 };
44
45   coords = Map[ {Re[#], Im[#]} & , newpoints, {2}];
46
47   lines = Map[ Line, Join[ coords, Transpose[ coords]] ];
48
49   Graphics[lines]
50 ]

```

---

```

53 CartesianMap[ func_, {x0_, x1_, dx_:Automatic}, {y0_, y1_, dy_:Automatic}, opts_] :=
54   Module[ {x,y,coords,plotpoints, ndx=dx, ndy=dy},
55     plotpoints= PlotPoints/.Options[Plot];
56   If[ dx==Automatic, ndx=N[(x1-x0)/(plotpoints-1)] ];
57   If[ dy==Automatic, ndy=N[(y1-y0)/(plotpoints-1)] ];
58   coords = Table[ N[ func[ x+I y]], {x,x0,x1,ndx},{y,y0,y1,ndy}];
59   Show[MakeLines[coords], opts, AspectRatio→Automatic, Axes→Automatic, ImageSize→Small];
60 ]

```

---

```

63 PolarMap[ func_, {r0_:0, r1_, dr_:Automatic}, {p0_, p1_, dp_:Automatic} ,opts_] :=
64   Module[ {r,p,coords,plotpoints, ndr=dr, ndp=dp},
65     plotpoints= PlotPoints/.Options[Plot];
66   If[ dr==Automatic, ndr=N[(r1-r0)/(plotpoints-1)] ];
67   If[ dp==Automatic, ndp=N[(p1-p0)/(plotpoints-1)] ];
68   coords = Table[ N[ func[ r Exp[ I p] ] ], {r,r0,r1,ndr},{p,p0,p1,ndp}];
69   Show[MakeLines[coords], opts, AspectRatio→Automatic, Axes→Automatic, ImageSize→Small];
70 ]

```

73 End[ ]

76 EndPackage[ ]

# Images

Many functions in *Mathematica* work on images.

You can get an image, for example, by copying or dragging it from the web or from a photo collection.

You can also just capture an image directly from your camera using the function `CurrentImage` (it works in browsers, on mobile devices, on PCs).

```
CurrentImage[]
```

```
imgDodecaedro = ;
```

```
(* pixel dimensions of an image *)
ImageDimensions[imgDodecaedro]
{320, 240}
```

What if I do not have a camera on my PC, or if I cannot use it?

Instead of `CurrentImage[]`, get a test image, e.g.:

```
imgTest = ExampleData[{"TestImage", "House2"}];
ImageDimensions[imgTest]
ImageResize[imgTest, {150, 150}]

? ExampleData

imgList = ExampleData["TestImage"];
(* n. test images for Image Processing *)
Print["In ExampleData[\"Test Images\"] there are ",
Length[imgList], " test images "];
(* the 7-th test image is "Apples" *)
(* the 24-th test image is "House2" *)
(* the 25-th test image is "JellyBeans" *)
{Part[imgList, 7], Part[imgList, 24], Part[imgList, 25]}
imgApples = ExampleData[{"TestImage", "Apples"}];
ImageResize[imgApples, {500, 300}]
(* imgJellyBeans=ExampleData[{"TestImage","JellyBeans"}];
ImageResize[imgJellyBeans,{150,150}] *)
```

## ColorNegate and ImageResize

You can apply functions to images just like you apply functions to numbers or lists or anything else.

`ColorNegate` (that we saw in connection with colors) also works on images, giving a "negative image".

```
(* Work on your previous test image, or copy and paste imgDodecaedro *)
width = 150;
height = 100;

(* ImageResize[ColorNegate[], {width, height}] ==
ImageResize[ColorNegate[imgDodecaedro], {width, height}] *)

GraphicsRow[{  

  imgDodecaedro,  

  ImageResize[imgDodecaedro, {width, height}],  

  ImageResize[ColorNegate[imgDodecaedro], {width, height}]}]
```

## Blur

The number in `Blur` is the range of pixels that get blurred together.

```
(* default*)
Blur[imgDodecaedro] == Blur[imgDodecaedro, 2]
GraphicsRow[{Blur[imgDodecaedro], Blur[imgDodecaedro, 10]}, ImageSize → Small]

(* Row puo' essere usato con qualsiasi lista di espressioni *)
Row[{Blur[imgDodecaedro], Blur[imgDodecaedro, 10]}]
Row[{Blur[imgDodecaedro], Blur[imgDodecaedro, 10]}, " "]

(* the argument of Spacer is in pt, i.e. points of a printer device *)
Row[{Blur[imgDodecaedro], Blur[imgDodecaedro, 10]}, Spacer[2]]

(* Invisible[] alike \phantom{ }. Here we
use it to insert the space taken by letter i *)
Row[{Blur[imgDodecaedro], Blur[imgDodecaedro, 10]}, Invisible[i]]

Row[Table[Blur[imgDodecaedro, n], {n, 0, 15, 5}]]
```

## ImageCollage puts images together:

```
ImageCollage[
Table[
  Blur[imgDodecaedro, n],
  {n, 0, 15, 5}],
ImageSize → Tiny]
```

```
(* Create a collage from a weighted list of images*)
b5 = Blur[imgDodecaedro, 5];
b15 = Blur[imgDodecaedro, 15];
b40 = Blur[imgDodecaedro, 40];

Row[{imgDodecaedro, b5, b15, b40}]

(* ImageCollage[ { w1→img1, w2→img2, ... } ]
   creates a collage of images img_i
   based on their corresponding weights w_i *)
ImageCollage[{  

  1 → imgDodecaedro,  

  4 → b5,  

  9 → b15,  

  1 → b40},  

 ImageSize → Small]
```

## Image Analysis

There is lots of **analysis** one can do on images.

E.g., DominantColors finds a list of the most important colors in an image

```
DominantColors[imgDodecaedro]
```

Binarize makes an image Black and White .

To decide what is Black and what is White, Binarize uses a threshold.

If you do not specify such a threshold, Binarize picks one based on analyzing the distribution of colors in the image.

```
width = 150;
height = 100;
imgDodecaedroBin = Binarize[imgDodecaedro];
ImageResize[imgDodecaedroBin, {width, height}]
DominantColors[imgDodecaedroBin]
```

## EdgeDetect

Another type of analysis is **edge detection** :

finding where in the image there are **sharp changes** in color.

```
width = 150;
height = 100;
imgDodecaedroEdge = EdgeDetect[imgDodecaedro];
ImageResize[imgDodecaedroEdge, {width, height}]
```

```
(* Add original image and its edge detection result *)
ImageResize[
  ImageAdd[imgDodecaedro, imgDodecaedroEdge],
  {width, height}]
```

It is often convenient to do image processing interactively, creating interfaces using **Manipulate**.

E.g., Binarize lets you specify a **threshold** (for what will be turned black as opposed to white): often the best way to find the right threshold is to interactively experiment with it.

```
width = 150;
height = 100;
Manipulate[
  ImageResize[
    Binarize[imgDodecaedro, t],
    {width, height}],
  {t, 0, 1}]

(* Manipulate[Binarize[imgTest,t],{t, 0, 1}] *)
```

## Vocabulary

CurrentImage[ img]	capture the current image from your PC, etc.
ColorNegate[ img]	negate the colors in an image
Binarize[ img]	convert an image to Black/White
Blur[ img , n]	blur an image
EdgeDetect[ img ]	detect the edges in an image
DominantColors[ img ]	get a list of dominant colors in an image
ImageCollage[{ img1 , img2 , img2}]	put together images in a collage
ImageAdd[ img1 , img2 ]	add color values of two images
ImageResize	
Row, Grid	

## Exercises

**10.1** ColorNegate the result of the EdgeDetect of an image (ImageResize).

**10.2** Use Manipulate to make an interface to Blur an image from 0 to 20.

**10.3** Make a Table of the results of EdgeDetect on a blurred image, with Blur from 1 to 9.

**10.4** Make an ImageCollage of an image and its Blur, EdgeDetect, Binarize.

**10.5** Add an image to a binarized version of it.

**10.6** Create a Manipulate to display Edges of an image, as it gets Blurred from 0 to 20.

**10.7** Image operations work on Graphics and Graphics3D. Edge detect a picture of a sphere.

**10.8** Make a Manipulate to make an interface to Blur a Purple pentagon from 0 to 20.

**10.9** Create a collage of 9 images of Disks, each with a Random Color (and a Yellow Background)

**10.10** Use ImageCollage to make a combined image of spheres with hues from 0 to 1 in steps of 0.2.

**10.11** Make a Table of Blurs of a Disk, by an amount from 0 to 30, in steps of 5.

**10.12** Use ImageAdd to add an image of a Disk to an image.

**10.13** Use ImageAdd to add an image of a Red octagon to an image.

**10.14** Add an image to its ColorNegate version of its EdgeDetect.

**+10.1** EdgeDetect a binarized image.

**+10.2** ColorNegate the EdgeDetect of an image of a regular pentagon.

**+10.3** Use ImageAdd to add an image to itself.

**+10.4** Use ImageAdd to add together the images of regular polygons with 3 to 100 sides.

## Tech Notes

- Images can appear directly in *Mathematica* code as a consequence of *Mathematica* being symbolic.
- A convenient way to get collections of images is to use WikipediaData :
- WebImageSearch gets images by searching the Web (see § 44)
- Many arithmetic operations work, on images, **directly pixel-by-pixel**  
⇒ you do not explicitly have to use **ImageAdd**, **ImageMultiply**, etc.

# Arrays, or Lists of Lists

Table can be used (further than to make lists) to create higher-dimensional arrays of values.

```
(* 4 copies of x *)
Table[x, 4]
Table[x, {4}];
{x, x, x, x}

(* 4 copies of the list of 5 elements {x,x,x,x,x} *)
Table[x, 4, 5]
{{x, x, x, x, x}, {x, x, x, x, x}, {x, x, x, x, x}, {x, x, x, x, x}}
```

Use **Grid** to display the result in a grid

```
Grid[Table[x, 4, 5]]
TableForm[Table[x, 4, 5]];
x x x x x
x x x x x
x x x x x
x x x x x
```

Table with 2 variables to make a 2D array.

The **first** variable corresponds to the **row**;  
the **second** to the **column**.

```
Table[red, {red, 0, 1, .2}]
{0., 0.2, 0.4, 0.6, 0.8, 1.}
```

```
(* Make an array of colors:
grid1 = Red going down, row by row,
blue going across, column by column *)
grid1 = Grid[
  Table[
    RGBColor[red, 0, blue],
    {red, 0, 1, .2},
    {blue, 0, 1, .2}
  ]];
grid2 = Grid[
  Table[
    RGBColor[red, 0, blue],
    {blue, 0, 1, .2},
    {red, 0, 1, .2}
  ]];
Row[{grid1, "      ", grid2}]






(* Show every array element as its row number *)
Grid[Table[i, {i, 3}, {j, 5}]]
1 1 1 1 1
2 2 2 2 2
3 3 3 3 3

(* Show every array element as its column number*)
Grid[Table[j, {i, 3}, {j, 5}]]
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5

(* Show every array element as its {row, column} number*)
Grid[Table[{i, j}, {i, 3}, {j, 5}]]
{1, 1} {1, 2} {1, 3} {1, 4} {1, 5}
{2, 1} {2, 2} {2, 3} {2, 4} {2, 5}
{3, 1} {3, 2} {3, 3} {3, 4} {3, 5}
```

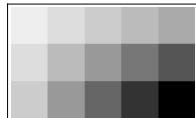
```
(* Generate an array in which each
element is the sum of its row and column number *)
Grid[Table[i+j, {i, 3}, {j, 5}]]
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8

(* Generate a multiplication table *)
Grid[Table[i*j, {i, 3}, {j, 5}]]
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
```

**ArrayPlot** lets you visualize values in an array.

Larger values are shown darker.

```
ArrayPlot[
  Table[i*j, {i, 3}, {j, 5}],
  ImageSize → Tiny]
```

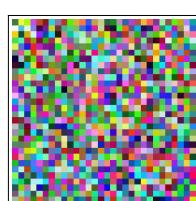
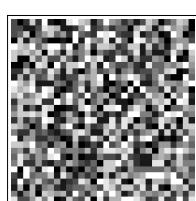


Generate and plot an array of random (integer or color) values:

```
SeedRandom[3];
bw = ArrayPlot[
  Table[
    RandomInteger[10], 30, 30],
  ImageSize → Tiny];

col = ArrayPlot[
  Table[
    RandomColor[], 30, 30],
  ImageSize → Tiny];

Row[{bw, "      ", col}]
```



Images are arrays of pixels.

Color images have each pixel with red, green and blue values.

Black/White images have pixels with values 0 (black) or 1 (white).

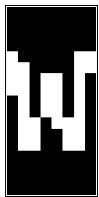
You can get the actual pixel values using `ImageData`.

```
(* Find the value of pixels in an image of a "W")
imgW = Rasterize["W"];
imgWbw = Binarize[imgW];
Row[{ImageResize[imgW, 40], "      ", ImageResize[imgWbw, 40]}]
imgWdata = ImageData[imgWbw]
```



```
{ {1, 1, 1, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 1, 1},
{1, 1, 1, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 1, 1}, {0, 1, 1, 1, 1, 1, 0}, {0,
0, 1, 1, 1, 1, 0}, {0, 0, 1, 0, 0, 1, 0}, {0, 0, 1, 0, 0, 1, 0}, {1, 0, 1, 0, 0, 1, 0}, {1, 0, 1, 0, 0, 1, 0}, {1, 0, 0, 1, 0, 0, 1}, {1, 0, 0, 1, 0, 0, 1}, {1, 0, 0, 1, 1, 0, 0}, {1, 0, 1, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 1, 1}, {1, 1, 1, 1, 1, 1, 1} }
```

```
(* Use ArrayPlot to visualize the array of values*)
ArrayPlot[imgWdata, ImageSize → Tiny]
```



The image has low resolution, because that is how `Rasterize` made it in this case.

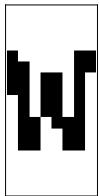
The image is White-on-Black (it is not Black-on-White), because, by default, `ArrayPlot` renders larger values in a darker shade (unlike in an image or in `RGBColor`, in which 0=Black, 1=White).

You can do arithmetic with arrays (= list of lists), just like with lists.

E.G., to swap 0 and 1, perform the subtraction **1 - everything** (so every 0 becomes  $1 - 0 = 1$ , and every 1 becomes  $1 - 1 = 0$ ).

```
(* Find pixel values, then do arithmetic to swap 0 and 1 in the array *)


```



## Vocabulary

Table[x,4,5]	make a 2D array of values
Grid[array]	lay out values from an array in a grid
ArrayPlot[array]	visualize the values in an array
ImageData[image]	get the array of pixel values from an image

## Exercises

**13.1** Make a  $12 \times 12$  multiplication table .

**13.2** Make a  $5 \times 5$  multiplication table for Roman numerals .

**13.3** Make a  $10 \times 10$  grid of random colors .

**13.4** Make a  $10 \times 10$  grid of randomly colored random integers between 0 and 10.

**13.5** Make a grid of all possible strings consisting of pairs of letters of the (English) alphabet (“aa”, “ab”, etc.) .

**13.6** Visualize  $\{1, 4, 3, 5, 2\}$  with a PieChart, NumberLine, LinePlot and BarChart. Place these in a  $2 \times 2$  grid.

**13.7** Make an ArrayPlot of Hue values  $x^*y$ , where  $x$  and  $y$  each run from 0 to 1 in steps of 0.05 .

**13.8** Make an ArrayPlot of Hue values  $x/y$ , where  $x$  and  $y$  each run from 1 to 50 in

steps of 1.

**13.9** Make an ArrayPlot of the Lengths of RomanNumeral Strings in a multiplication table up to  $100 \times 100$ .

**+13.1** Make a  $20 \times 20$  addition table.

**+13.2** Make a  $10 \times 10$  grid of randomly colored random integers between 0 and 10, that have random size up to 32.

---

## Q & A

Can the limits of one variable in a table depend on another?

Yes, later ones can depend on earlier ones.

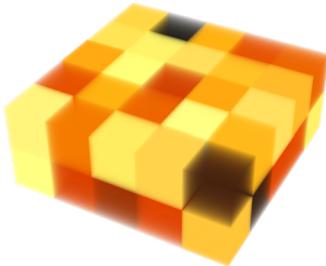
```
Grid[
  Table[x,
    {i, 4}, {j, i}]
]
x
x x
x x x
x x x x
```

Can I make tables that are lists of lists of lists?

Yes, you can make tables of any dimension.

Image3D gives a way to visualize 3D arrays.

```
(* RandomReal[range, {n1, n2, ...}] yields an array of n1 x n2 x ....
pseudo-random reals *)
(* ImageData3D[data] represents a 3D
image with pixel values given by the array data *)
SeedRandom[3];
myimg3d = Image3D[
  RandomReal[1, {2, 5, 5}]
]
```



Why does 0 correspond to black, and 1 to white, in images?

0 means zero intensity of light, i.e. Black.

1 means maximum intensity, i.e. White.

How do I get the original image back from the output of `ImageData`?

Just apply the function `Image` (or `Image3D`) to it.

```
SeedRandom[3];
myimg3d = Image3D[RandomReal[1, {2, 5, 5}]];
idata = ImageData[myimg3d];
Image3D[%]
```



## Tech Notes

- Arrays in *Mathematica* are just lists in which each element is itself a list.  
*Mathematica* also allows more general structures, that mix lists and other things.
- mondimensional Lists in *Mathematica* correspond to mathematical vectors; lists of equal-length

sublists correspond to matrices.

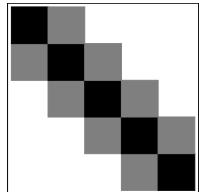
- If most of the entries in an array are 0 (or some other fixed value), you can use **SparseArray** to construct an array by giving positions and values of nonzero elements.

```
(* Construct a tridiagonal matrix using patterns for indices *)
```

```
s = SparseArray[  
{  
 {i_, i_} → -2,  
 {i_, j_} /; Abs[i - j] == 1 → 1  
 },  
 {5, 5}]
```

```
SparseArray[  Specified elements: 13  
 Dimensions: {5, 5} ]
```

```
ArrayPlot[s, ImageSize → Tiny]
```



# Strings and Text

*Mathematica* allows computing with text.

You enter text as a string, indicated by quotes (" ... ">,

A string is an atom.

There exist, basically, 3 types of **atom**: symbol, number, string (of characters).

■ **Symbol**: sequence of letters and/or numbers and/or the character \$ ; a symbol cannot start with a number.

■ There are 4 types of numbers.

▪ **Integer**, exact (a sequence of decimal digits *dddddd*);

▪ **Rational**, exact (it has the form Integer1/Integer2, reduced to lowest terms);

▪ **Real**, approximate, with any specified precision (it has the form *ddd.ddd* i.e. it is a variable-precision floating-point);

▪ **Complex** (it has the form *a+b\*I* , where a, b can be any of the previous number types).

■ **String** (of characters): any sequence of characters in between open/closed double quotes " ... "

Just like when you enter a number, a string (on its own) comes back unchanged, except that the quotes are not visible, when the string is displayed in output.

```
"This is a string."
```

```
AtomQ[%]
```

This is a string.

True

There are many functions that work on strings.

**StringLength, StringReverse**

**ToUpperCase**

**StringTake, StringJoin**

**Characters** breaks a string into a list of its characters

Sometimes it is useful to turn strings into lists of their characters.

Each Character is a string of Length 1.

```
Characters["A string is made of characters"]
```

```
{A, , s, t, r, i, n, g, , i, s, , m, a, d, e, , o, f, , c, h, a, r, a, c, t, e, r, s}
```

Once a string is broken into a list of characters, all the usual built-in (for Lists) can be used on it.

```
cc = Characters["An apple"];
scc = Sort[cc]
FullForm[scc]
(* TableForm[{  
    InputForm[scc]],  
    FullForm[scc],  
    OutputForm[scc]  
}] *)
{ , a, A, e, l, n, p, p}

List[" ", "a", "A", "e", "l", "n", "p", "p"]
```

## TextWords, TextSentences

Functions like **StringJoin**, or **Characters**, work on strings of any kind (strings that do not need to be meaningful).

Other functions are useful on meaningful text (written, say, in English).

**TextWords**, for instance, gives a list of the words in a string of text:

```
oneSentence = "This is a sentence: sentences  
    are made of phrases, and phrases are made of words.";  
tw = TextWords[oneSentence]
(* Length of each word in the list tw *)StringLength[tw]
{This, is, a, sentence, sentences, are,  
    made, of, phrases, and, phrases, are, made, of, words}

{4, 2, 1, 8, 9, 3, 4, 2, 7, 3, 7, 3, 4, 2, 5}
```

**TextSentences** breaks a text string into a list of sentences:

```
oneSentence =
"This is a sentence: sentences are made of phrases; phrases are made of words.";
fewSentences =
"This is a sentence. Sentences are made of phrases. Phrases are made of words.";
(* A seconda della punteggiatura, ho 1 frase oppure piu' frasi *)
ts0 = TextSentences[oneSentence]
ts = TextSentences[fewSentences]
(* Length of each sentence *)
StringLength[ts0]
StringLength[ts]

{This is a sentence: sentences are made of phrases; phrases are made of words.}
{This is a sentence., Sentences are made of phrases., Phrases are made of words.}
```

{77}

{19, 30, 26}

# WikipediaData , WordCloud

There are many ways to get text into *Mathematica*.

One example is the **WikipediaData** function, which gets the current text of Wikipedia articles.

(\* Wikipedia article on “computers” \*)

```
wdPC = WikipediaData["computers"];
```

(\* Get the first 113 characters of such article \*)

```
PC113 = StringTake[wdPC, 113]
```

A computer is a digital electronic machine that can be programmed to carry out sequences of arithmetic or logical

(\* wdTS contains the first 3 sentences of the wdPC article \*)

```
wdTS = TextSentences[wdPC, 3]
```

(\* 2nd element of wdTS, ie the 2nd sentence of the wdPC article \*)

Part[wdTS, 2]

{A computer is a digital electronic machine that can be programmed to carry out sequences of arithmetic or logical operations (computation) automatically., Modern computers can perform generic sets of operations known as programs., These programs enable computers to perform a wide range of tasks.}

Modern computers can perform generic sets of operations known as programs.

(\* Lets check \*)

**Hyperlink["https://en.wikipedia.org/wiki/Computer", Appearance → "DialogBox"]**

<https://en.wikipedia.org/wiki/Computer>

Given a piece of text, to get a sense of its content, a convenient way is to create a **WordCloud**.

```
WordCloud[wdPC, ImageSize -> Tiny]
```

(\* not surprisingly,

“computer” and “computers” are the most common words in the article \*)



# WordList

*Mathematica* has lots of built-in knowledge about words (in English and other languages).

**WordList** gives lists of words.

```
(* Get the first 5 words from a list of common English words *)
wlEnglish = WordList[];
Length[wlEnglish]
Take[wlEnglish, 5]

39176

{a, aah, aardvark, aback, abacus}
```

Note: aardvark = formichiere  
 to shift aback = tirare indietro  
 to take aback = prendere alla sprovvista

```
(* Get the last 5 words from a list of common Italian words *)
wlIta = WordList[Language → "Italian"];
Length[wlIta]
Take[wlIta, -5]

116854

{zuppe, zuppi, zuppiera, zuppo, zizzarellone}
```

Let us create a **WordCloud** from the first letters of all the words

```
(* wlEnglish=WordList[]; *)
wlst1 = StringTake[wlEnglish, 1];
WordCloud[
  wlst1,
  FontFamily → "Arial",
  ImageSize → Tiny]
```



```
Length[$FontFamilies]
2104
```

```
WordCloud[
wlst1,
FontFamily → "Comic Sans MS",
ImageSize → Tiny]
```



RomanNumeral

IntegerName

Alphabet, LetterNumber, Transliterate

Rasterize

If you want to, you can also turn text into images, which you can then manipulate using image processing.

Rasterize makes a raster, or bitmap, that is an image gets described in pixels.

Generate an image of a piece of text.

```
imgRaster = Rasterize[ Style["ABC", 50] ]
```

Do image processing on it:

```
EdgeDetect[imgRaster]
```



## Vocabulary

“string”	a string
StringLength[“string”]	length of a string
StringReverse[“string”]	reverse a string
StringTake[“string”,4]	take first 4 characters of a string
StringJoin[“string”, “string”]	join strings together
StringJoin[{“string”, “string”}]	join a list of strings

ToUpperCase["string"]	convert characters to uppercase
Characters["string"]	convert a string to a list of characters
TextWords["string"]	list of words from a string
TextSentences["string"]	list of sentences
WikipediaData["topic"]	Wikipedia article about a topic
WordCloud["text"]	word cloud based on word frequencies
WordList[ ]	list of common words in English
Alphabet[]	list of letters of the alphabet
LetterNumber["c"]	where a letter appears in the alphabet
FromLetterNumber[n]	letter appearing at position n in the alphabet
Transliterate["text"]	transliterate text into English
Transliterate["text", "alphabet"]	transliterate text into other alphabets
RomanNumeral[n]	convert a number to its Roman numeral string
IntegerName[n]	convert a number to its English name string
InputForm["string"]	show a string with quotes
Rasterize["string"]	make a bitmap image

## Exercises

- 11.1** Join two copies of the string “Hello”.
- 11.2** Make a single string of the (English) Alphabet, in Uppercase.
- 11.3** Generate a String of the (English) Alphabet in Reverse order.
- 11.4** Join 10 copies of the String “MatComp”.
- 11.5** Use StringTake, StringJoin and Alphabet to get “abcdef” .
- 11.6** Create a Column with increasing numbers of letters from the string “this is about strings” (IntegerPart).
- 11.7** Make a BarChart of the Lengths of the words in “A long time ago, in a galaxy far, far away”.
- 11.8** Find the StringLength of the Wikipedia article for “computer”.
- 11.9** Find how many words are in the Wikipedia article for “computer”.
- 11.10** Find the first Sentence in the Wikipedia article about “strings”.
- 11.11** Make a String from the first letters of all Sentences in the Wikipedia article

about computers.

**11.12** Find the Max word-length among (English) words from WordList[ ].

**11.13** Count the number of words in WordList[ ] that start with “q”.

**11.14** Make a ListLinePlot of the Lengths of the first 100 words from WordList[ ].

**11.15** Use Characters and StringJoin to make a WordCloud of all letters in the words from WordList[ ].

**11.16\*** Use StringReverse to make a WordCloud of the last letters in the words from WordList[ ].

**11.17** Find the Roman numerals for the year 1959.

**11.18** Find the Maximum StringLength of any Roman-numeral year from 1 to 2021.

**11.19** Make a WordCloud from the **first** characters of the Roman numerals up to 10.

**11.20** Use Length to find the Length of English, Russian, Italian Alphabets.

**11.21** Generate the Uppercase Greek Alphabet.

**11.22** Make a BarChart of the Letter Numbers in “compute”.

**11.23** Use FromLetterNumber to make a String of 10 Random letters (SeedRandom[25]).

**11.24** Make a list of 10 Random 3-letter Strings (SeedRandom[3]).

**11.25** Transliterate “Your name” into Greek.

**11.26** Get the Arabic Alphabet and transliterate it into English.

**11.27** Make a White-on-Black size-100 letter “A”.

**11.28** Use Manipulate to make an interactive selector of characters (of font size-20) from the Alphabet .

**11.29** Use Manipulate to make an interactive selector of Black-on-White outlines of Rasterized size-50 characters from the Alphabet, controlled by a drop-down menu.

**11.30** Use Manipulate to create a “vision simulator” that Blurs a size-50 letter “A” by an amount from 0 to 20.

**+11.1** Generate a String of the Alphabet followed by the Alphabet written in reverse.

**+11.2** Make a Column of a string of the Alphabet and its Reverse.

**+11.3** Find how many sentences are in the Wikipedia article for “computer”.

**+11.4** Join together without spaces, the words in the first sentence in the Wikipedia article for “strings”.

**+11.5** Find the Length of the **longest** word in the Wikipedia article about computers.

**+11.6** Plot the lengths of Roman numerals for numbers up to 200.

**+11.7** Generate a string by joining the Roman numerals up to 10.

**+11.8** Make a ListLinePlot of the successive letter numbers for the concatenation of all Roman numerals up to 10.

**+11.9** Find the Maximum StringLength of the Name of any Integer up to 10.

**+11.10** Make a list of UpperCase size-20 letters of the Alphabet in RandomColor.

**+11.11** Make a list of 10 random 3-letter strings with the Italian alphabet.

**+11.12** Create a Manipulate to display Edges in size-100 letter A, Blurred from 0 to 30.

**+11.13** Add together White-on-Black size-100 letters A and B.

## Q & A

## Tech Notes

## More to Explore

Guide to String Manipulation in *Mathematica*

# Sound

In *Mathematica*, sound works a lot like **Graphics**, but instead of having things like **Circle[ ]**, one has **SoundNote**.

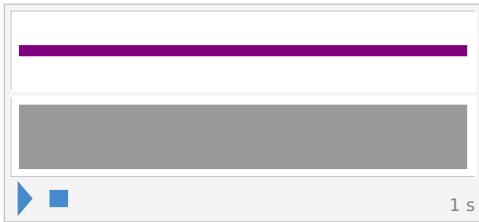
Press the play button  to actually play sounds.

By default, notes sound as if played on a Piano;

the default **duration** of each note is 1 second;

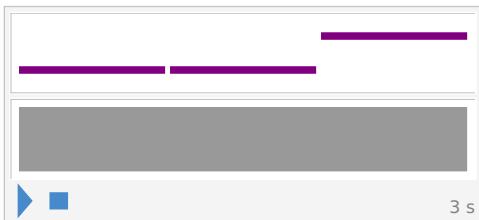
the default note is “middle C” (i.e. “DO”).

```
Sound[ SoundNote[] ]
(* Sound[ SoundNote[ "C" ], 1, "Piano" ] *)
```



You can specify a sequence of notes by giving them in a list.

```
(* Sound[ { SoundNote["C"], SoundNote["C"], SoundNote["G"] } ] *)
note3 = Map[
  SoundNote, {"C", "C", "G"}
];
Sound[note3]
```



To specify the pitch (altezza → nota acuta|grave; frequenza fondamentale dell’onda sonora) of a note, you can also use a number.

```
(* Grandezze fondamentali di una nota: altezza,
intensità o volume, timbro o purezza *)
```

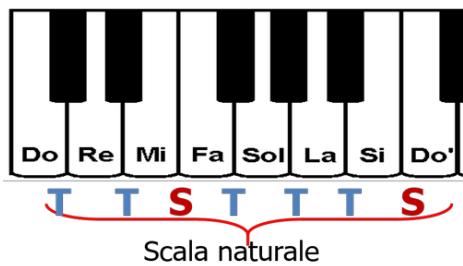
<http://www3.unisi.it/ricerca/prog/musica/linguaggio/suono.htm>

DO (DO centrale, middle C, “C”, “C4”) is 0.

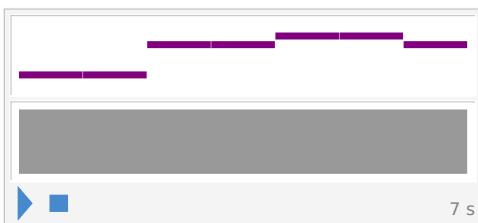
Each semitone above the DO goes up by 1.

SOL (SOL centrale, middle G, “G”, “G4”) is 7, since it is 7 semitones above the DO.

```
nd = NotebookDirectory[];
pathOCTAVE = StringJoin[nd, "tono-e-semitono.png"];
imgOCTAVE = Import[pathOCTAVE];
ImageDimensions[imgOCTAVE]
ImageResize[imgOCTAVE, {500, 300}]
{815, 502}
```



```
note3num = Map[
  SoundNote, {0, 0, 7, 7, 9, 9, 7}
];
(* DO, DO, SOL, SOL, LA, LA, SOL *)
(* note3num=Map[SoundNote,{"C","C","G","G","A","A","G"}]; *)
Sound[note3num]
```

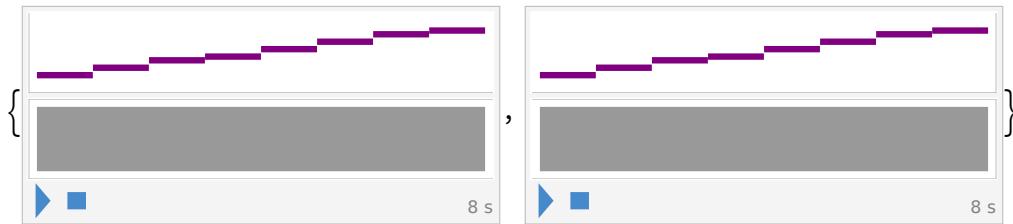


An octave (8 subsequent notes) is 12 semitones.

```
nd = NotebookDirectory[];
pathNOTE = StringJoin[nd, "12NOTEnotazioneanglosassone.jpg"];
imgNOTE = Import[pathNOTE];
ImageDimensions[imgNOTE]
ImageResize[imgNOTE, {250, 100}]
{430, 177}
```



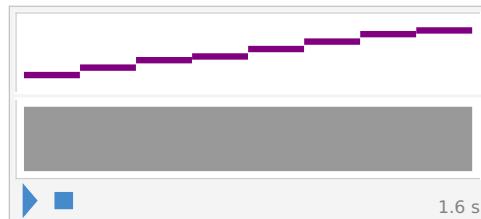
```
note8 = Map[SoundNote, {"C", "D", "E", "F", "G", "A", "B", "C5"}];
note8num = Map[SoundNote, {0, 2, 4, 5, 7, 9, 11, 12}];
{Sound[note8], Sound[note8num]}
```



By default, each note lasts 1 second.

Use **SoundNote[pitch, length]** to get a different length.

```
myLen = 0.2;
(* DO/"C"/0, RE/"D"/2, MI/"E"/4, FA/"F"/5,
   SOL/"G"/7, LA/"A"/9, SI/"B"/11, DO/"C5"/12 *)
(* la prima Table crea {0,2,4} *)
(* la seconda Table crea {5,7,9,11} *)
(* Join crea la lista {0,2,4, 5,7,9,11, 12} *)
notes = Join[
  Table[k, {k, 0, 4, 2}],
  Table[k, {k, 5, 11, 2}],
  {12}];
Sound[
  Map[ SoundNote[#, myLen] &, notes]
]
```



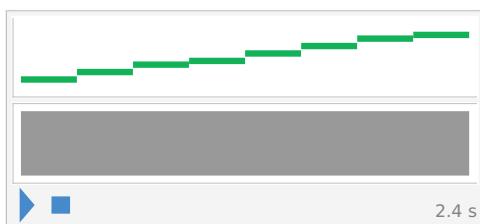
In addition to the Piano, SoundNote can handle various instruments.

The name of each instrument is a string.

```

myLen = 0.3;
myInstrument = "Guitar";
(* DO/"C"/0, RE/"D"/2, MI/"E"/4, FA/"F"/5,
SOL/"G"/7, LA/"A"/9, SI/"B"/11, DO/"C5"/12 *)
notes = Join[
  Table[k, {k, 0, 4, 2}],
  Table[k, {k, 5, 11, 2}],
  {12}];
Sound[
  Map[ SoundNote[#, myLen, myInstrument] &, notes]
]

```



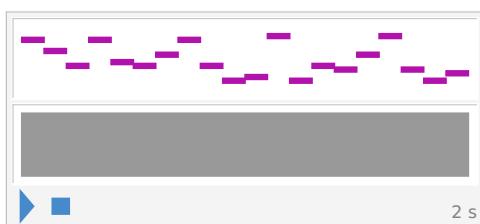
Make "random music" (different every time you generate it).

E.G. Play a sequence of 20 notes, with length 0.1 and random pitches from 0 to 12:

```

SeedRandom[2];
Sound[
  Table[
    SoundNote[ RandomInteger[12], 0.1, "Violin" ],
    20]
]

```



## Vocabulary

Sound[...]	create a sound from notes
SoundNote["C"]	a named note
SoundNote[5]	a note with a numbered pitch
SoundNote[5, 0.1]	a note played for a specified time
SoundNote[5, 0.1, "Guitar"]	a note played on a certain instrument
Map	

---

## Exercises

- 12.1** Generate the sequence of notes with pitches 0, 4, 7 .
- 12.2** Generate 2 seconds of playing middle A on a **Cello** .
- 12.3** Generate a “riff” (short musical phrase) of notes, from pitch 0 to 48 in steps of 1, with each note lasting 0.05 seconds .
- 12.4** Generate a sequence of notes, from pitch 12 down to 0 in steps of 1 (total duration 1.3 sec, i.e. each note lasts 0.1 sec).
- 12.5** Generate a sequence of 5 notes , starting with DO (0/middle C) and going up by an octave (12 semi-tones) at a time.
- 12.6** Generate a sequence of 10 notes on a **Trumpet** , with random pitches from 0 to 12 (SeedRandom[8]) and duration 0.2 seconds .
- 12.7** Generate a sequence of 10 notes, with random pitches from 1 to 12 (SeedRandom[11]), and random durations from 1/10 to 10/10 (tenths of a second).
- 12.8** Create a Sound from Notes with pitches given by the Integer Digits of  $2^{31}$ , each playing for 0.1 seconds
- 12.9** Create a Sound from Notes with pitches given by the Characters in ACCADDE, each playing for 0.3 seconds, sounding like a (violon)Cello.
- 12.10** Create a Sound from Notes with pitches given by the LetterNumber of characters in your Name or Surname, each playing for 0.1 seconds, sounding like a Harp.
- +12.1** Generate a sequence of 3 notes of 1 second of playing middle D, on **Cello**, **Piano**, **Guitar** .
- +12.2** Generate a sequence of notes, from pitch 0 to 12, going up in steps of 3.
- +12.3** Generate a sequence of 5 notes, starting with middle C, then successively going up by a perfect fifth (7 semitones, quinta perfetta/giusta) at a time .
- +12.4** Generate 0.02-second notes, with pitches given by the StringLengths of the

## Names of Integers from 1 to 200 (Module).

---

### Q & A

#### How do I know which instruments are available?

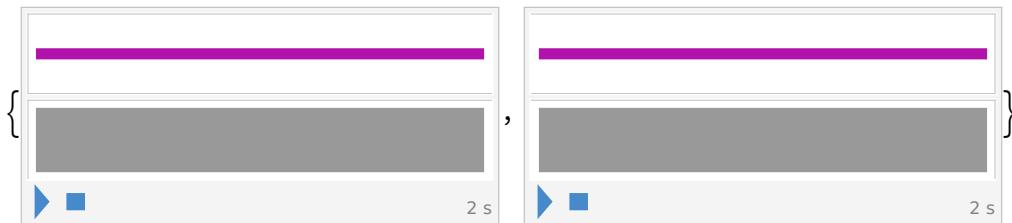
Look at "Details and Options" in the **SoundNote** Help page  
(or just start typing and see the completions you are offered).

All the standard MIDI (Musical Instrument Digital Interface) instruments are available, including percussion.

- You can also use instrument numbers from 1 to 128.

In *Mathematica*, the first MIDI instrument (Piano) is 1, rather than 0.

```
(* {Sound[SoundNote["A", 2, 1]], Sound[SoundNote["A", 2, "Piano"]]} *)
(* {Sound[SoundNote["A", 2, 25]], Sound[SoundNote["A", 2, "Guitar"]]} *)
{Sound[SoundNote["A", 2, 41]], Sound[SoundNote["A", 2, "Violin"]]}
(* {Sound[SoundNote["A", 2, 43]], Sound[SoundNote["A", 2, "Cello"]]} *)
```

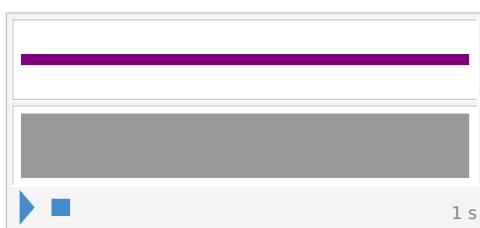


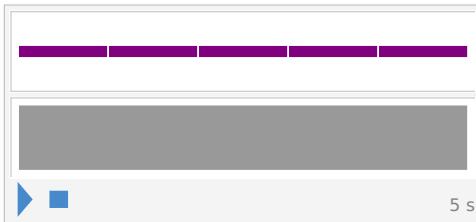
⇒ Styles from 1 to 128, representing MIDI **instruments**:

"Accordion"	"Agogo"	"AltoSax"	"Applause"
"Atmosphere"	"Bagpipe"	"Bandoneon"	"Banjo"
"BaritoneSax"	"Bass"	"BassAndLead"	"Bassoon"
"Bird"	"BlownBottle"	"Bowed"	"BrassSection"
"Breath"	"Brightness"	"BrightPiano"	"Calliope"
"Celesta"	"Cello"	"Charang"	"Chiff"
"Choir"	"Clarinet"	"Clavi"	"Contrabass"
"Crystal"	"DrawbarOrgan"	"Dulcimer"	"Echoes"
"ElectricBass"	"ElectricGrandPiano"	"ElectricGuitar"	"ElectricPiano"
"ElectricPiano2"	"EnglishHorn"	"Fiddle"	"Fifths"
"Flute"	"FrenchHorn"	"FretlessBass"	"FretNoise"
"Glockenspiel"	"Goblins"	"Guitar"	"GuitarDistorted"
"GuitarHarmonics"	"GuitarMuted"	"GuitarOverdriven"	"Gunshot"
"Halo"	"Harmonica"	"Harp"	"Harpsichord"
"Helicopter"	"HonkyTonkPiano"	"JazzGuitar"	"Kalimba"
"Koto"	"Marimba"	"MelodicTom"	"Metallic"
"MusicBox"	"MutedTrumpet"	"NewAge"	"Oboe"
"Ocarina"	"OrchestraHit"	"Organ"	"PanFlute"
"PercussiveOrgan"	"Piano"	"Piccolo"	"PickedBass"
"PizzicatoStrings"	"Polysynth"	"Rain"	"Recorder"
"ReedOrgan"	"ReverseCymbal"	"RockOrgan"	"Sawtooth"
"SciFi"	"Seashore"	"Shakuhachi"	"Shamisen"
"Shanai"	"Sitar"	"SlapBass"	"SlapBass2"
"SopranoSax"	"Soundtrack"	"Square"	"Steeldrums"
"SteelGuitar"	"Strings"	"Strings2"	"Sweep"
"SynthBass"	"SynthBass2"	"SynthBrass"	"SynthBrass2"
"SynthDrum"	"SynthStrings"	"SynthStrings2"	"SynthVoice"
"Taiko"	"Telephone"	"TenorSax"	"Timpani"
"Tinklebell"	"TremoloStrings"	"Trombone"	"Trumpet"
"Tuba"	"TubularBells"	"Vibraphone"	"Viola"
"Violin"	"Voice"	"VoiceAahs"	"VoiceOohs"
"Warm"	"Whistle"	"Woodblock"	"Xylophone"

■ A percussion event is specified with no pitch

```
Sound[SoundNote["JingleBell"]]
(* tamburello=SoundNote["Tambourine"];
Sound[ConstantArray[ tamburello,3]] *)
metronomo = SoundNote["MetronomeClick"];
Sound[ ConstantArray[metronomo, 5] ]
```





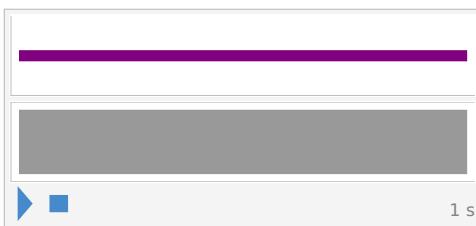
⇒ Possible **percussions** events :

```
"BassDrum"      "BassDrum2"     "BellTree"      "Cabasa"
"Castanets"     "ChineseCymbal" "Clap"        "Claves"
"Cowbell"        "CrashCymbal"   "CrashCymbal2" "ElectricSnare"
"GuiroLong"      "GuiroShort"   "HighAgogo"    "HighBongo"
"HighCongaMute"  "HighCongaOpen" "HighFloorTom" "HighTimbale"
"HighTom"         "HighWoodblock" "HiHatClosed"  "HiHatOpen"
"HiHatPedal"    "JingleBell"   "LowAgogo"    "LowBongo"
"LowConga"       "LowFloorTom"  "LowTimbale"   "LowTom"
"LowWoodblock"   "Maracas"     "MetronomeBell" "MetronomeClick"
"MidTom"          "MidTom2"     "MuteCuica"   "MuteSurdo"
"MuteTriangle"   "OpenCuica"   "OpenSurdo"   "OpenTriangle"
"RideBell"        "RideCymbal"  "RideCymbal2" "ScratchPull"
"ScratchPush"    "Shaker"      "SideStick"   "Slap"
"Snare"           "SplashCymbal" "SquareClick" "Sticks"
"Tambourine"     "Vibraslap"   "WhistleLong" "WhistleShort"
```

## How do I play notes below DO (middle C)?

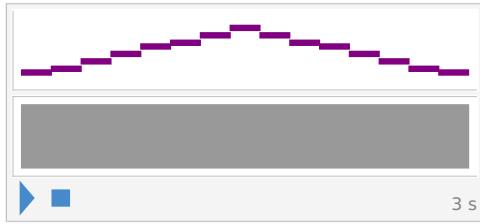
Use negative numbers.

`Sound[ SoundNote[-10] ]`



```
(* Posso usare indici negativi per formare una scala ascendente e discendente *)
myLen = 0.2;
(* Form andata = {-12,-11,-9,-7,-5,-4,-2,0}  *)
andata = Union[
  (* Form {0,-2,-4} *) Table[k, {k, 0, -4, -2}],
  (* Form {-5,-7,-9,-11} *) Table[k, {k, -5, -11, -2}],
  {-12}
];
(* Form ar={-12,-11,-9,-7,-5,-4,-2,0,-2,-4,-5,-7,-9,-11,-12} *)
ar = Join[
  Drop[andata, -1],
  Reverse[andata]
];

Sound[
  Map[ SoundNote[#, myLen] &, ar ]
]
```



## What are sharp (#) and flat (b) notes called?

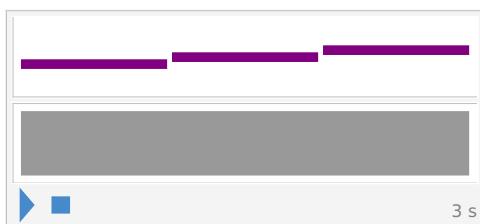
E # (E sharp/diesis), Ab (A flat/bemolle), etc.

They also have numbers (e.g. E # is 5 ).

Symbols “#” and “b” can be typed as ordinary keyboard characters (# and ₪ characters are available too).

```
(* MI/"E"/4 *)
Sound[{
```

```
  SoundNote["Eb"], (* SoundNote[3],*)
  SoundNote["E"] , (* SoundNote[4], *)
  SoundNote["E#"] (* SoundNote[5] *)
}]
```



## How do I make a chord (accordo)?

Put the names of the Notes in a list

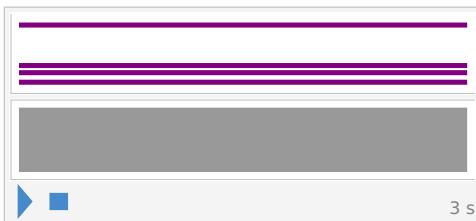
```
(* { RE, FA diesis, LA, RE di 2 ottave più alto del RE centrale *)
```

```
Sound[
```

```
  SoundNote[{"D", "F#", "A", "D6"}, 3]
```

```
]
```

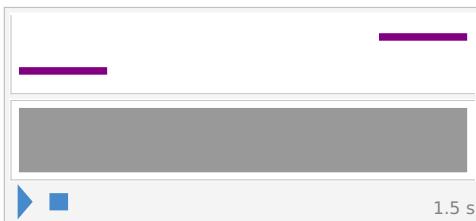
```
(* Sound[ SoundNote[{2,6,9,26}] ] *)
```



## How do I make a rest? How do I overlap notes?

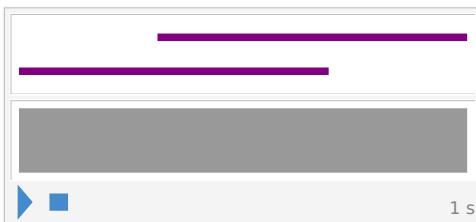
- Use **None** as duration

```
Sound[{  
  SoundNote["C", 0.3],  
  SoundNote[None, 0.9],  
  SoundNote["G", 0.3]  
}]
```



- It is also possible to overlap notes.

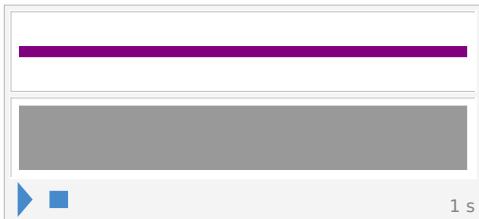
```
Sound[{  
  SoundNote["C", {0, 0.7}],  
  SoundNote["G", {0.3, 1}]\n}]
```



How do I get a sound to play immediately, without having to press the play button (EXAMPLES)?

■ Use **EmitSound**

```
ssn = Sound[ SoundNote[] ]
```



```
EmitSound[ ssn ]
```

```
(* Button creates a button *)
Button[ "play", EmitSound[ ssn ] ]
```



■ Another example (Play[])

■ A more complex example (phone keypad) - Play[], Outer[], MapIndexed[]

Why do I need quotes in the name of a note like RE (“D”) ?

Because Built-in functions to manipulate Sound expect a string (or a number).

Moreover, typing D (instead of “D”) would be interpreted as a (built-in) function named D[ ].

```
D[y x^2, x]
```

```
2 x y
```

Can I record audio and manipulate it (AudioCapture)?

## Tech Notes

SoundNote is MIDI sound (Musical Instrument Digital Interface) . There are other “sampled sound”, e.g., ListPlay or Audio.

To get spoken output, use **Speak**. To make a beep, use **Beep**.

```
(* Does not work under Linux *) Beep[]
```

```
Button["press me", Speak["Perfect"]]
```



## More to Explore

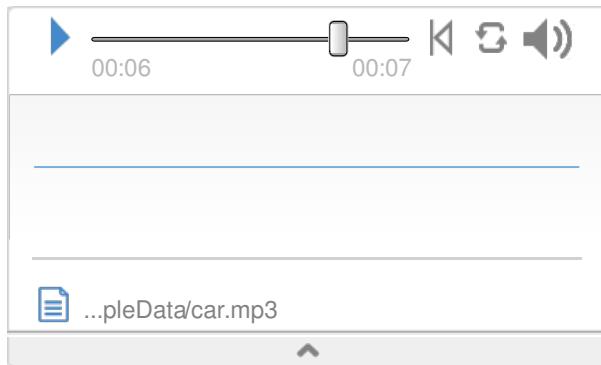
Guide to **Sound Generation** in the *Mathematica*

<https://reference.wolfram.com/language/guide/SoundAndSonification.html>

Import ed Export di file audio

<https://www.wolfram.com/mathematica/new-in-10/enhanced-sound-and-signal-processing/import-and-export-mp3-files.html>

`Import["ExampleData/car.mp3"]`



# Tests and Conditionals

Test whether  $2 + 2$  is equal to 4 :

**2 + 2 == 4**

True

Test whether  $2 \times 2$  is greater than 5 :

**2 + 2 > 5**

False

## If

The conditional function **If[ test, x, y, z ]** gives:

“x” then-branch (*test* is True) ,

“y” else-branch (*test* is False),

“z” otherwise-branch (*test* is neither True nor False, i.e., Indeterminate).

```
(* In statement 1, test is True => result is x *)
If[ 2 + 2 == 4, x, y, z]
x

(* Unequal != ,typed as "!=" *)
(* In statement 2, test is False => result is y *)
If[2 + 2 != 4, x, y, z]
y

(* Since a,b are not Set, in statement 3,
test is neither True nor False => result is z *)If[a < b, x, y, z]
z
```

## If and TrueQ

```
(* TrueQ forces the condition to return a Boolean value *)
TrueQ[a < b]
(* Even with a,b not Set, test TrueQ[a<b] is False => result is y *)
If[TrueQ[a < b], x, y, z]
False

y
```

## If and Map

By using **Map** (/@), we can apply **If** (as a pure function) to every element of a list.

```
(* Map a test on a list *)
mylist = {1, 2, 3, 4, 5, 6, 7, 8, 9};
Map[ If[# < 4, x, y] & , mylist ]
(* equivalently *)
If[# < 4, x, y] & /@ mylist;

(* another test on mylist *)
Map[ If[# != 4, x, y] & , mylist]
{x, x, x, y, y, y, y, y, y}
{x, x, x, y, x, x, x, x, x}
```

## Select

It is often useful to Select elements in a list that satisfy a test.

You can do this with **Select**, giving your test as a pure function.

```
(* Select from a list those element that verify a test *)
mylist = {1, 2, 3, 4, 5, 6, 7, 8, 9};
Select[ mylist , # > 3 & ]
Select[ mylist , 2 <= # <= 5 & ]
{4, 5, 6, 7, 8, 9}
{2, 3, 4, 5}
```

## Select and EvenQ, IntegerQ, PrimeQ, ....

Beyond size comparisons like <, >, ==, *Mathematica* includes many other kinds of tests:

**EvenQ** , **OddQ** , **IntegerQ** , **PrimeQ** ....

(“Q” indicates that the functions are asking a question.)

```
{ EvenQ[4], EvenQ[5], OddQ[4], OddQ[5] }
{True, False, False, True}

(* Select Even numbers/Primes from a list *)
mylist = {1, 2, 3, 4, 5, 6, 7, 8, 9};
Select[mylist, EvenQ]
Select[mylist, PrimeQ]
{2, 4, 6, 8}
{2, 3, 5, 7}
```

```
(* Select Integers from a list *)
mylist2 = {1., 2., 3., 4., 5, 6., 7, 8, 9};
Select[mylist2, IntegerQ]
(* Forsee the output of the following Select *)
Select[mylist2, EvenQ];
Select[mylist2, PrimeQ];
{5, 7, 8, 9}
```

## Select with AND, OR, NOT

To combine tests:

&& represents And

|| represents Or

! represents Not

```
mylist = {1, 2, 3, 4, 5, 6, 7, 8, 9};
(* Select Even elements greater than 2 *)
Select[mylist, EvenQ[#]&& # > 2 & ]
{4, 6, 8}

(* Select elements that are NOT either "Even" OR ">4"
   i.e. Select Odd elements ≤ 4 *)
s1 = Select[mylist, !(EvenQ[#] || # > 4) &]
s2 = Select[mylist, OddQ[#]&& # ≤ 4 &];
s1 == s2
{1, 3}
```

True

## Select , LetterQ, LetterNumber

There are many other "Q functions" that ask various kinds of questions.

**LetterQ** tests whether a string consists entirely of letters.

```
(* The space between letters is not a letter; nor is "!" *)
Map[ LetterQ, {"ab", "a b", "!", "2", "$", "#", "@", "%", "&", "+"} ]
{True, False, False, False, False, False, False, False, False}

(* Turn a string into a list of Characters, then test which are letters *)
char = Characters["30 is the best!"]
Map[LetterQ, char]
(* Select the Characters that are letters *)
Select[char, LetterQ]
{3, 0, , i, s, , t, h, e, , b, e, s, t, !}
```

```

{False, False, False, True, True, False,
 True, True, True, False, True, True, True, False}

{i, s, t, h, e, b, e, s, t}

(* Task: Select letters that appear after Position 13, strictly,
in the UK and Italian Alphabets : "o" is at Position 15 and 13, respectively *)
char = Characters["our test"];
Select[char, LetterQ[#] && LetterNumber[#] > 13 & ]
Select[char, LetterQ[#] && LetterNumber[#, "Italian"] > 13 &]
{o, u, r, t, s, t}
{u, r, t, s, t}

```

## Select and WordList

You can use **Select** to find words, in (English and) Italian, that are palindromes.

```

Select[WordList[], StringReverse[#] == # &];
Select[WordList[Language → "Italian"], StringReverse[#] == # &]
{aerea, afa, ala, alla, ama, ara, atta, avallava, aveva, ebbe, elle,
 ere, esse, ingegni, inni, ivi, non, onorarono, oro, oso, osso, otto}

```

## Select and MemberQ

**MemberQ** tests whether something appears as an element, or member of a list.

```

MemberQ[{1, 3, 5, 7}, 5]
True

(* Select pairs containing x *)
Select[
 { {1, y}, {2, x}, {3, z}, {x, 5}, {y, 6}, {z, 7} },
 MemberQ[#, x] &
]
(* Select entries containing x *)
Select[
 { {1, y}, {2, x}, {3, z}, {x, 5}, {z, 6},
   {1, y, 4}, {2, x, y}, {z, 2, y},
   {4, 1, x, t}, {x, 1, x, t} },
 MemberQ[#, x] &
]
{{2, x}, {x, 5}}
{{2, x}, {x, 5}, {2, x, y}, {4, 1, x, t}, {x, 1, x, t}}

```

## SelectFirst

```
(* Select numbers in the Range 50 to 100, whose digit sequences contain 2 *)
Select[
  Range[50, 100],
  MemberQ[ IntegerDigits[#, 2] &
]

(* Equivalently, in this particular case *)
Select[
  Range[50, 100],
  Last[ IntegerDigits[#]] == 2 &
];
{52, 62, 72, 82, 92}

(* Select the First of the numbers in the Range 50 to 100,
whose digit sequences contain 2 *)
SelectFirst[
  Range[50, 100],
  MemberQ[IntegerDigits[#, 2] &
]
52
```

## Cases

```
(* Equivalently, with Cases,
form all even numbers in [50,100], then Take the second one *)
even = Cases[
  Range[50, 100],
  _?EvenQ
]
(* The construction pattern?test applies a function test to the whole
expression matched by pattern, to determine whether there is a match *)
Take[even, {2}]
First[%]
{50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70, 72,
 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100}

{52}

52
```

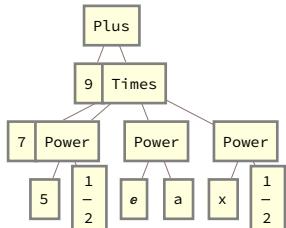
`Cases[{ $e_1, e_2, \dots$ }, pattern]` gives a list of those  $e_i$  that match the **pattern**

The first argument to **Cases** does not need to have List as Head (default level is 1)

```
expr = 7 Sqrt[5 x] Exp[a] + 9
```

```
TreeForm[expr]
```

$$9 + 7 \sqrt{5} e^a \sqrt{x}$$



```
(* default level is 1 *) Cases[ expr , _Integer ]
(* all levels *) Cases[ expr , _Integer, Infinity ]
(* level 3 only *) Cases[ expr , _Integer, {3}]
(* levels from 1 to 2 included *) Cases[ expr , _Integer, 2]
```

```
{9}
```

```
{9, 7, 5}
```

```
{5}
```

```
{9, 7}
```

## ImageInstanceQ and Entity

- **ImageInstanceQ** is a function based on machine-learning, that tests whether an Image is an Instance of a particular kind of something (e.g., a cat).

**ImageInstanceQ[ image , category ]** tests whether **image** is an instance of **category**

```
(* Test if an image is that of a cat *)
```

```
ImageInstanceQ[, Entity["Species", "Species:FelisCatus"]]
```

True

- **Entity** is a symbolic expression, built-in, in the sense that *Mathematica* knows about thousands of types of real-world entities:  
Cities, Countries, Chemicals, Movies, Populations, Satellites, Species, Airports Companies , etc.

```
Entity["City", List["Rome", "Lazio", "Italy"]]
```

```
Rome
```

- To enter an **Entity** in input, use **CTRL+=** (Shift +=, under Cloud), that yields a box in which it is possible to type in English (e.g. capital of Italy) then evaluate the cell

The screenshot shows a Mathematica input cell. Inside the cell, the word "Italy" is highlighted with an orange border, indicating it has been selected as an Entity. To its right, the text "COUNTRY" is displayed in a smaller font. To the right of the entity name is a yellow rectangular input field containing the text "capital city". Above this input field are two small buttons: one with three dots and another with a checkmark.

Rome

**FullForm[%]**

```
Entity["City", List["Rome", "Lazio", "Italy"]]
```

- Examples with Species (cats).

**Entity["Species", "Species:FelisCatus"]**

domestic cat

## Geographic examples with Select, Entity, **Position**, GeoDistance

Here is a geographic example of Select :

find which cities, in a list, are **less** than 3000 miles from San Francisco.

```
(* Do not put a space in compound names *)
(* city, area/region, state *)
testCity = Entity["City", {"SanFrancisco", "California", "UnitedStates"}];
worldCities = {
  Entity["City", {"London", "GreaterLondon", "UnitedKingdom"}],
  Entity["City", {"Madrid", "Madrid", "Spain"}],
  Entity["City", {"Tokyo", "Tokyo", "Japan"}],
  Entity["City", {"Chicago", "Illinois", "UnitedStates"}]
};

Select[
 worldCities,
 GeoDistance[#, testCity] < Quantity[3000, "Miles"] &
]
{Chicago}
```

```

italianCities20list = {
    {"Sulmona", "Abruzzes"}, 
    {"Monopoli", "Apulia"}, 
    {"Rotonda", "Basilicata"}, 
    {"Tropea", "Calabria"}, 
    {"Caserta", "Campania"}, 
    {"Bologna", "EmiliaRomagna"}, 
    {"Pordenone", "FriuliVeneziaGiulia"}, 
    {"Rome", "Lazio"}, 
    {"Alassio", "Liguria"}, 
    {"Sondrio", "Lombardy"}, 
    {"Urbino", "Marche"}, 
    {"Termoli", "Molise"}, 
    {"Ivrea", "Piemonte"}, 
    {"Olbia", "Sardegna"}, 
    {"PiazzaArmerina", "Sicily"}, 
    {"Volterra", "Toscana"}, 
    {"Pinzolo", "TrentinoAltoAdige"}, 
    {"Gubbio", "Umbria"}, 
    {"Aosta", "ValleDAosta"}, 
    {"Garda", "Veneto"} 
};

italianCities20 = Map[
    Entity["City", Flatten[List[#, "Italy"]]] &,
    italianCities20list]
{Sulmona, Monopoli, Rotonda, Tropea, Caserta, Bologna, 
Pordenone, Rome, Alassio, Sondrio, Urbino, Termoli, Ivrea, 
Olbia, Piazza Armerina, Volterra, Pinzolo, Gubbio, Aosta, Garda}

```

→ **Position**[*expression, pattern*] gives a list of the Positions at which objects matching *pattern* appear in *expression*.

```

testRome = Entity["City", {"Rome", "Lazio", "Italy"}];
positionRome = Flatten[
    Position[ italianCities20, testRome ]
]
{8}

(* Drop Rome from the list of Italian Cities *)
italianCities19 = Drop[italianCities20, positionRome];

```

```
(* Find which Italian cities, in our current list, are <200km from Rome *)
testRome = Entity["City", {"Rome", "Lazio", "Italy"}];
Select[
  italianCities19,
  GeoDistance[#, testRome] < Quantity[200, "Kilometer"] &
]
{Sulmona, Caserta, Gubbio}
```

---

## Vocabulary

**a==b** test for equality  
**a**<**b** tests whether less  
**a**>**b** tests whether greater  
**a**≤**b** tests whether less or equal  
**a**≥**b** tests whether greater or equal  
**If[test, u, v]** gives u if test is True, and v if False  
**Select[list,f]** selects elements that pass a test  
**EvenQ[x]** tests whether even  
**OddQ[x]** tests whether odd  
**IntegerQ[x]** tests whether an integer  
**PrimeQ[x]** tests whether a prime number  
**LetterQ[string]** tests whether there are only letters  
**MemberQ[list,x]** tests whether x is a member of list  
**ImageInstanceQ[image,category]** tests whether image is an instance of category  
**Entity**  
**Cases**  
**Position**

---

## Exercises

- 28.1** Test whether 123^321 is greater than 456^123.
- 28.2** Get a list of numbers up to 100 whose IntegerDigits Total up to be < 5.
- 28.3** Make a list of the first 20 integers, where Prime numbers are Styled Red.
- 28.4** Find words in WordList[ ] that begin and end with the letter “p”.
- 28.4bis** Find words in WordList[ ], in Italian, that begin and end with the letter

“n”.

**28.5** Make a list of the first 100 Primes, keeping only those whose last IntegerDigit is < 3 (Array).

**28.6** Find RomanNumerals up to 50 that do **not** contain “I”.

**28.7** Get a list of RomanNumerals up to 50 that are palindromes (and have more than 1 letter).

**28.8** Find Names of Integers up to 100 that begin and end with the same letter.

**28.8bis** Find Italian Names of Integers up to 100 that begin and end with the same letter.

**28.9** Get a list of Words, longer than 18 Characters, from the Wikipedia article on “words”.

**28.10** Starting from 1000, divide by 2 if the number is Even, compute  $3 \# + 1$  & if the number is Odd; do this 111 times (Collatz problem 1937 : this process converges to 1, regardless of which positive integer is chosen initially, with an appropriate number of iterations. Named after the German mathematician Lothar Collatz, 1910-1990).

**28.11** Make a WordCloud of 5-letter words in the Wikipedia article on computers.

**28.12** Find words in WordList[ ], whose first 3 letters are the same as their last 3 read backward, and that are not palindrome.

**28.13** Find all 12-letter words in WordList[ ] for which the Total of LetterNumber values is 100.

**+28.1** Make a Table of Integers up to 25, where every integer ending in 3 is replaced with 0.

**+28.2** Use Table and If to make a  $5 \times 5$  array that is 1 on its leading diagonal, and 0 otherwise.

**+28.3** Get a list of numbers up to 1000 that are equal to 1 Mod 7 and also equal to 1 Mod 8.

**+28.4\*** Make a list of numbers up to 50, where multiples of 3 are replaced by Black ■, multiples of 5 by White □, and multiples of 3 and 5 by Red ■ (swatch).

**+28.5** Use Select to get a list of planets whose mass is larger than Earth.

**+28.6** Make a  $50 \times 50$  **ArrayPlot**, in which a pixel (square) at position  $(i, j)$  is Black if  $\text{Mod}[i, j] == 0$ , otherwise is White.

**+28.7** Make a  $100 \times 100$  ArrayPlot, in which a pixel (square) is Black if the values of both its x and y positions do not contain a 5.

---

## Q & A

**When do I need to use parentheses with `&&`, `||`, etc.?**

There is an order of operations that is an analog of Arithmetic.

`And[]` `&&` is like `Times`

`Or[]` `||` is like `Plus`

`Not[]` `!` is like Subtract or Minus, i.e.,  $-x$  is `Times[-1, x]`

Furthermore, the Laws of De Morgan (British mathematician, 1806-1871) are valid:

`Not(A And B)` is `(Not A) Or (Not B)`

`Not(A Or B)` is `(Not A) And (Not B)`.

If in doubt, use **LogicalExpand**

```
(* This means (NOT p)AND q *)
!p && q ;
LogicalExpand[%]
q &&!p

(* This means NOT(p AND q)
which is (NOT P)OR(NOT q) *)
!(p && q);
LogicalExpand[%]
!p || !q
```

**What are some other “Q” query functions?**

`NumberQ`, `StringContainsQ`, `BusinessDayQ` and `ConnectedGraphQ` are a few.

Are there other ways to find real-world entities with certain properties than using `Select`?

Yes.

E.G., use `Entity["Country", "Population" -> GreaterThan[ num ] ]` to find "implicit entities".

Then, use `EntityList` to get explicit lists of entities.

```
countriesHighlyPopulated = Entity["Country", "Population" → GreaterThan[60 * 10 ^ 6]]
```

```
EntityList[countriesHighlyPopulated]
```

```
{China, India, United States, Indonesia, Pakistan,
Brazil, Nigeria, Bangladesh, Russia, Mexico, Japan, Ethiopia,
Philippines, Egypt, Vietnam, Democratic Republic of the Congo,
Turkey, Iran, Germany, Thailand, United Kingdom, France, Italy}
```

## Tech Notes

- True and False are symbols (they are **not** represented by 1 and 0) and are called Booleans (after the English mathematician George Boole, 1815-1864).
- Expressions with `&&`, `||`, etc. are called Boolean expressions.
- In *Mathematica* :
  - x is a symbol (see § 33) that can represent anything ;

`x == 1` is an equation, that is not immediately True or False;

`x === 1` tests whether x is the Same as 1 : since it is not, it gives False.

## More to Explore

[Guide to Functions That Apply Tests in \*Mathematica\*](#)

<https://reference.wolfram.com/language/guide/TestingExpressions.html>

[Guide to Boolean Computation in \*Mathematica\*](#)

<https://reference.wolfram.com/language/guide/BooleanComputation.html>

# More about Patterns

- \_ ("blank") stands for anything
- x\_** ("x blank") stands for anything, but calls it x
- \_h** stands for anything with head h
- x\_h** stands for anything with head h, and calls it x

► Define a function whose argument is an Integer, named **n**:

```
digitback[n_Integer] := Framed[Reverse[IntegerDigits[n]]]
(* Framed[expr] displays a framed version of expr *)
```

Function **digitback[n]** evaluates whenever the argument is a n Integer, with positive or negative sign (Note that `IntegerDigits[-1]` yields `{1}` )

```
testlist = {1234, -6712, x, {4, 3, 2}, y /. y → 20, z → 3, 2^32, 22., 0};
Flatten[{  
  Map[digitback, testlist],  
  digitback[2, 2],  
  digitback[]  
}]
{{4, 3, 2, 1}, {2, 1, 7, 6}, digitback[x], digitback[{4, 3, 2}], {0, 2}, digitback[z → 3],  
{6, 9, 2, 7, 6, 9, 4, 9, 2, 4}, digitback[22.], {0}, digitback[2, 2], digitback[]}
```

► Sometimes you may want to put a **Condition** (`/;`) on a pattern.

For example, `n_Integer /; n > 0` means “any integer that is greater than 0”

Define a function whose argument is an Integer named **n**, with **n > 0**:

```
pdigitback[n_Integer /; n > 0] := Framed[Reverse[IntegerDigits[n]]]

testlist = {1234, -6712, x, {4, 3, 2}, y /. y → -20, z → 3, 2^32, 22., 0};
Flatten[{  
  Map[pdigitback, testlist],  
  pdigitback[2, 2],  
  pdigitback[]  
}]
{4, 3, 2, 1}, pdigitback[-6712], pdigitback[x], pdigitback[{4, 3, 2}],  
pdigitback[-20], pdigitback[z → 3], {6, 9, 2, 7, 6, 9, 4, 9, 2, 4},  
pdigitback[22.], pdigitback[0], pdigitback[2, 2], pdigitback[]}
```

- The **Condition** can go (almost) anywhere—even at the end of the whole definition.

For example, define different cases of the **check** function:

```
Clear[x, y];  
check[x_, y_] := Red /; x > y  
check[x_, y_] := Green /; x ≤ y  
{check[1, 2],  
 check[2, 1],  
 check[3, 4],  
 check[50, 60],  
 check[60, 50]}
```

(\* Alternative \*)

```

Clear[x, y];
check2[x_, y_] /; x > y := Red
check2[x_, y_] /; x ≤ y := Green
{check2[1, 2],
 check2[2, 1],
 check2[3, 4],
 check2[50, 60],
 check2[60, 50]}

{Green, Red, Green, Green, Red}

```

**WARNING !**

## **From Specific to General**

- ▷ Double blank    stands for any sequence of one or more arguments.
  - ▷ Triple blank    stands for zero or more.
  - ▷ For example, define a function that looks for Black and White (in this order) in a list, and picks out the shortest sequence s between the first-met Black and its first subsequent White.
  - ▷ Pattern {    , Black , s    , White ,    } matches Black followed by White, with any elements before/between/after them:

```

testcolors = {Yellow, White, Black, Blue, Black, Green, Orange, White, Purple, White,
             Black, Yellow, White, Black, Green, Yellow, White, Red, White, Blue, Brown};
bw[{___, Black, ___, White, ___}] := {s};
testcolors
bw[testcolors]

{Yellow, White, Black, Blue, Black, Green, Orange, White, Purple, White,
             Black, Yellow, White, Black, Green, Yellow, White, Red, White, Blue, Brown}
{Black, Yellow, White, Black, Green, Yellow, White, Red, White, Blue, Brown}

```

- **bwex** picks the **Longest** sequence between the first Black and the last White:

```

bwex[___, Black, Longest[s___], White, ___] := {s};
testcolors
bwex[testcolors]
{Yellow, White, Black, Blue, Black, Green, Orange, White, Purple, White, Black, Yellow, White, Black, Green, Yellow, White, Red, White, Blue, Brown}
{Blue, Black, Green, Orange, White, Purple, White, Black, Yellow, White, Black, Green, Yellow, White, Red}

► bwcut cuts out the longest run containing only Black|White,
and returns everything before the first Black|White
and everything after the last Black|White, separated by Red .

x|y|z  matches x or y or z
x..  matches any number of repetitions of x

bwcut[{a___, Longest[(Black | White) ..], b___}] := {{a}, Red, {b}};
testcolors2 =
{Gray, Gray, White, Black, White, Black, White, White, Black, Black, Yellow};
testcolors2
bwcut[testcolors2]
(* testcolors
  bwcut[testcolors] *)
{Gray, Gray, White, Black, White, Black, White, White, Black, Black, Yellow}
{{Gray, Gray}, Red, {Yellow}}

► The pattern x_ is actually short for
x:_
which means "match anything (i.e. _) and name the result x "
You can use notations like x:_ for more complicated patterns too .

► Define a pattern, named m, that matches a 2x2 matrix :

grid22[ m:{_, _, _, _}, {_, _, _} ] := Grid[m, Frame → All];
test = {
{{a, b}, {c, d}},
{{12, 34}, {56, 78}} ,
{123, 456} ,
{{1, 2, 3}, {4, 5, 6}}
};

Map[grid22, test]
{{a|b, 12|34}, {c|d, 56|78}, grid22[{123, 456}], grid22[{{1, 2, 3}, {4, 5, 6}}]}

► Name the sequence r of Black|White, so it can be used in the result ( e.g. Length[{r}] ):
```

```
bwcut2[{a___, r : Longest[Black | White] .., b___}] := {{a}, Framed[Length[{r}]], {b}};
testcolors2
bwcut2[testcolors2]
{█, █, □, █, □, █, □, □, □, █, █, █, █}
{█, █, █, 8, {█}}
```

- Let us use patterns to implement the classic Computer Science **algorithm of sorting a list**, by repeatedly swapping pairs of successive elements that are found to be out of order. We write each step in the algorithm as a replacement for a pattern.

```
(* Swap the first pair b, a of elements
   found to be out of order *)
testlist0 = {5, 4, 1, 3, 2};
pattern = {x___, b_, a_, y___} /; b > a → {x, a, b, y};
ReplaceAll[testlist0, pattern]
{4, 5, 1, 3, 2}

(* Do the same operation 8 times, in this example,
eventually sorting the list completely *)
testlist = {4, 5, 1, 3, 2};
pattern = {x___, b_, a_, y___} /; b > a → {x, a, b, y};
nl = NestList[
  ReplaceAll[#, pattern] &, (* # /. pattern &,*)
  testlist,
  8]
{{4, 5, 1, 3, 2}, {4, 1, 5, 3, 2}, {1, 4, 5, 3, 2}, {1, 4, 3, 5, 2},
 {1, 3, 4, 5, 2}, {1, 3, 4, 2, 5}, {1, 3, 2, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}}
```

- We do not know how long this method will take to finish sorting a particular list. Therefore, use **FixedPointList**, which is like **NestList**, except that you do not have to tell a specific number of steps: it just goes on, until the result reaches a fixed point, where nothing more is changing.

```
(* Do the operation until a fixed point is reached *)
testlist = {4, 5, 1, 3, 2};
pattern = {x___, b_, a_, y___} /; b > a → {x, a, b, y};
fpl = FixedPointList[
  ReplaceAll[#, pattern] &, (* # /. pattern &,*)
  testlist]
{{4, 5, 1, 3, 2}, {4, 1, 5, 3, 2}, {1, 4, 5, 3, 2}, {1, 4, 3, 5, 2},
 {1, 3, 4, 5, 2}, {1, 3, 4, 2, 5}, {1, 3, 2, 4, 5}, {1, 2, 3, 4, 5}, {1, 2, 3, 4, 5}}
```

Use **Transpose**, to form columns of current positions of each element to be sorted, at each of the step taken.

**ListPlot** shows how the sorting process proceeds:

from step 1 to step 2, the green line (1) and the orange line (5) swap;

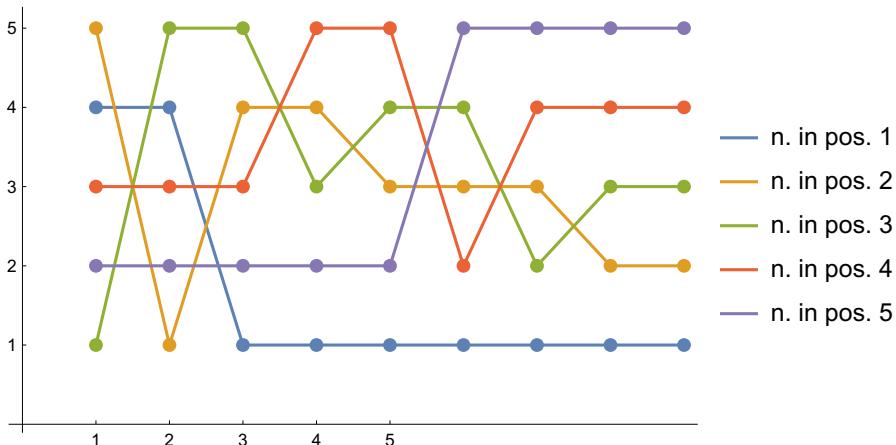
from step 2 to step 3, the blue line (4) and the orange line (5) swap;

and so on.

From step 8 to step 9, no swap occurs, therefore the fixed point is reached.

```
tfpl = Transpose[fpl]; TableForm[tfpl]
dims = Dimensions[tfpl];
lines = ListPlot[tfpl, Joined → True,
  PlotLegends → {"n. in pos. 1",
    "n. in pos. 2",
    "n. in pos. 3",
    "n. in pos. 4",
    "n. in pos. 5"}];
points = ListPlot[tfpl, PlotStyle → PointSize[0.02]];
Show[points, lines, Ticks → Range[dims]]
```

4	4	1	1	1	1	1	1	1	1
5	1	4	4	3	3	3	2	2	
1	5	5	3	4	4	2	3	3	
3	3	3	5	5	2	4	4	4	
2	2	2	2	2	5	5	5	5	

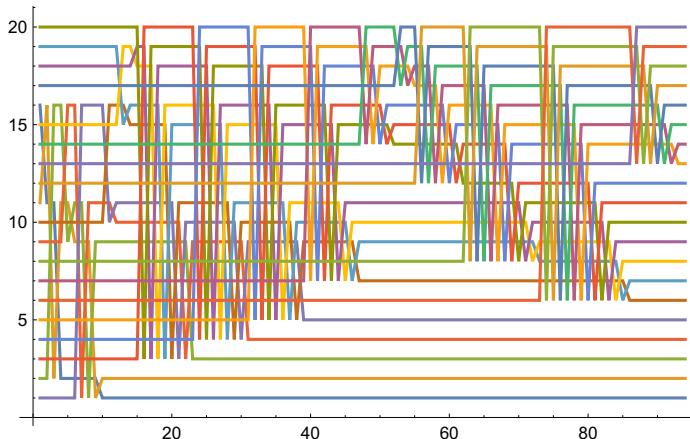


Sort a list of 20 elements obtained by randomly sampling Range[20].

```

SeedRandom[3];
testlistR = RandomSample[Range[20]];
pattern = {x___, b_, a___, y___} /; b > a → {x, a, b, y}; fplR = FixedPointList[
  ReplaceAll[#, pattern] &,
  testlistR
];
tfplR = Transpose[fplR];
dimsR = Dimensions[tfplR];
linesR = ListPlot[tfplR, Joined → True]
{16, 11, 2, 9, 1, 10, 19, 15, 18, 20, 3, 4, 5, 7, 14, 17, 12, 8, 6, 13}
{20, 94}

```



## Vocabulary

patt /; cond	a pattern that matches if a <b>Condition</b> is met
___	a pattern for any sequence of zero or more elements (“triple blank”)
patt..	a pattern for one or more repeats of <b>patt</b>
Longest[patt]	a pattern that picks out the longest sequence that matches
FixedPointList[f,x]	keep nesting f until the result no longer changes
RandomSample	
DownValues	

## Exercises

**41.1** In the Squares of the first 50 Integers, find those containing 2 successive repeated Digits.

**41.2** In the first 100 Roman Numerals, find those containing L, I, X, in this order.

**41.3** Define a function **f** that tests whether a non-empty List of Integers is the same as its Reverse.

**41.4** Obtain a List of pairs of successive words, in the Wikipedia article on alliteration, that have identical first letters and such a first letter is a vowel: show its first, penultimate and 20th elements .

**41.5** Use **Grid** to show the sorting process, seen in this section for {4, 5, 1, 3, 2}, with successive steps going down the page.

**41.6** Use **ArrayPlot** to show the sorting process, seen in this section for a list of length 20, with successive steps going across the page.

**41.7** Start with 1.0, then repeatedly apply  $(1/2)(\# + (2/\#)) \&$  until the result no longer changes.

**41.8** Implement the GCD algorithm by Euclid, in which {**a**, **b**} is repeatedly Replaced by {**b**, Mod[**a**, **b**] } until **b** is 0. Apply it to 12345, 54321.

**41.9** Define **Combinators**, using the Rules  $s[x_][y_][z_] \rightarrow x[z][y[z]]$ ,  $k[x_][y_] \rightarrow x$ . Then, starting with  $s[s][k][s[s][s]][s][s]$  , generate all possible combinations, applying the Rules until nothing changes. Give the last result.

**41.10** Remove all trailing 0's from the Digit List for Factorial[20].

**41.11** Start from {1, 0}. For 10 steps, repeatedly Remove the first 2 elements (from the current list) and: Append {0, 1} if the first removed element was 1; Append {1, 0, 0} if the first removed element was 0 . Obtain a list of the Lengths of the sequences produced (tag system).

**41.12** Start from {0, 0}. For 200 steps, repeatedly Remove the first 2 elements and: Append {2, 1} if the first removed element was 0, Append{0} if the first removed element was 1, Append {0, 2, 1, 2} if the first removed element was 2. Make a LinePlot of the Lengths of the sequences produced (tag system).

---

## Q & A

### What are other pattern constructs in *Mathematica*?

**Except[patt]** matches anything except patt.

```
(* This gives all elements except 0 *)
Clear[test]; test = {1, 0, 2, 0, 3};
Cases[test, Except[0]]

{1, 2, 3}

(* This gives all elements except Integers *)
Clear[test]; test = {a, b, 0, 1, 2, x, y};
Cases[test, Except[_Integer]]

{a, b, x, y}

(* Delete non-vowel characters from a string *)
vowels = Characters["aeiou"];
FullForm[vowels]
StringReplace["patterns guru", Except[vowels] → ""]

List["a", "e", "i", "o", "u"]

aeuu

(* Delete characters that are not Digits from a string *)
Clear[test]; test = "a0bb11ccc2d3e4444fffff5g";
(* DigitCharacter matches any character for which DigitQ gives True *)
StringReplace[test, Except[DigitCharacter] → ""]

0112344445
```

**PatternSequence[patt]** matches a sequence of arguments in a function.

```
(* Replace a strict subsequence in a list*)
myrule = {x __, PatternSequence[c, d, c], y __} → {x, Red, y};
mylist = {a, b, c, d, c, d, a, b, d};
ReplaceAll[mylist, myrule]
(* mylistR={a,b,c,d,c,d,a,b,d,c,d,c,c};*
ReplaceRepeated[mylistR, myrule]; *)
{a, b, █, d, a, b, d}

(* Match a periodic pattern *)
myrulep = {PatternSequence[x __, y __] ..} → x + y;
mylist2 = {a, b, a, b, a, b, a, b, a, b}; ReplaceAll[mylist2, myrulep]
(* mylist3={a,b,a,b,a,b,a,b,a,b,z};*
ReplaceAll[mylist3, myrulep] *)
```

a + b

**OrderlessPatternSequence**[patt] matches them in any order.

```
(* Match elements of a list in any order*)
SeedRandom[3];
mytest = RandomSample[Range[10]];
mypattern = {OrderlessPatternSequence[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]};
MatchQ[mytest, mypattern]
{8, 9, 2, 3, 1, 6, 7, 10, 4, 5}
True
```

Use OrderlessPatternSequence to make some of the function arguments orderless :

```
g[i_Integer, OrderlessPatternSequence[r_Real, s_String]] := {i, r, s}
g[1, 26.2, "string"]
g[1, "string", 26.2]
{1, 26.2, string}
{1, 26.2, string}

(* Only the last two arguments of g are orderless *)
g[26.2, 1, "string"]
g[1, 1., 1]
g[26.2, 1, string]
g[1, 1., 1]
```

**f[x\_:v]** defines v as a default value, so **f[]** is matched, with x being v.

```
f[z_ : 3] := 1 + z;
{f[] (* 1+3 *),
 f[1](* 1+1 *),
 f[4, 5](* f not a function of 2 variables *)}
{4, 2, f[4, 5]}

g[x_, z_ : 3] := x + z;
{g[] (* g not defined as a function of no variables *),
 g[1](* 1+3 *),
 g[4, 5](* 4+5 *)}
{g[], 4, 9}

h[x_ : 0, z_ : 3] := x + z;
{h[] (* 0+3 *),
 h[1] (* 1+3 *),
 h[4, 5](* 4+5 *)}
```

```

{3, 4, 9}

k[z_ : 3, x_ : 0] := x + z;
{k[], (*3+0*),
 k[1] (* 1+0 *),
 k[4, 5] (* 4+5 *)}

{3, 1, 9}

w[x_ : 0, y_, z_ : 0] := x + y + z;
{w[] (* w not defined as a function of no variables *),
 w[5] (* 0+5+0 *),
 w[4, 5] (* 4+5+0 *),
 w[4, 5, 6] (* 4+5+6 *)}

{w[], 5, 9, 15}

```

## How can one see all the ways a pattern could match a particular expression?

Use **ReplaceList**.

Replace gives the first match;

ReplaceList gives a list of all of them.

```

Clear[a, b, c, d, e, f];
test = {a, b, c, d, e, f};
pattern = {x_, y_} → {{x}, {y}};
Replace[test, pattern]
ReplaceList[test, pattern]

{{a}, {b, c, d, e, f}}

{{{a}, {b, c, d, e, f}}, {{a, b}, {c, d, e, f}},
 {{a, b, c}, {d, e, f}}, {{a, b, c, d}, {e, f}}, {{a, b, c, d, e}, {f}}}

```

## What does **FixedPointList** do if there is no fixed point?

It will eventually stop.

There is an option that tells it how far to go.

FixedPointList[f, x, n] stops after at most n steps.

```

FixedPointList[ Floor[#/2] &, 1000]
FixedPointList[ Floor[#/2] &, 1000, 4]

{1000, 500, 250, 125, 62, 31, 15, 7, 3, 1, 0, 0}

{1000, 500, 250, 125, 62}

```

---

# Tech Notes

## Repeated

In a repeating pattern patt .., you must use parentheses () or leave a space before the digit, to avoid confusion with decimal numbers.

```
rparenthesis = {(1.) ..} → x; (* OK *)
rspace = {1. ..} → x; (* OK *)
rule0 = {1 ...} → x; (* NO,
rule0 intended as {(1)...}→ x , where 1 is integer *)
testReal = {{1., 1.}, {1.}, {2, 1.}};
ReplaceAll[ testReal, rparenthesis]
ReplaceAll[ testReal, rspace]
ReplaceAll[ testReal, rule0]
{x, x, {2, 1.}}
{x, x, {2, 1.}}
{{1., 1.}, {1.}, {2, 1.}}

(* testint={{1,1},{1},{2,1}};
ReplaceAll[ testint,rparenthesis]
ReplaceAll[ testint,rspace]
ReplaceAll[ testint,rule0 ] *)
```

## Attributes

Functions can have **Attributes** that affect how pattern matching works.

For example, **Plus** has attributes Flat (associativity) and Orderless (commutativity).

### Attributes[Plus]

```
{Flat, Listable, NumericFunction, OneIdentity, Orderless, Protected}
```

Flat means that  $b+c$  can be pulled out of  $a+b+c+d$ .

It corresponds to the mathematical property of **associativity**.

If a symbol **f** has attribute Flat, then **f[f[a,b], f[c]]** is automatically reduced to **f[a,b,c]**.

```
ClearAll[f];
SetAttributes[f, Flat];
f[ f[a, b], f[c] ]
f[a, b, c]
```

```
(* Flat allows the pattern matcher to use associativity *)
ReplaceAll[ a+b+c+d , x_+y_ → {x, y} ]
{a, b+c+d}

ReplaceAll[ a+b+x+c+d , x__+b+y__+c+z__ → {x+y+z, b+c} ]
{a+d+x, b+c}
```

**Orderless** means that elements can be reordered, so  $a+c$  can be pulled out.

It corresponds to the mathematical property of **commutativity**.

Functions with attribute Orderless use canonical order .

```
(* Plus sorts its arguments because it has attribute Orderless *)
b+a
a+b
```

Any orderless function will sort its arguments :

```
ClearAll[g];
SetAttributes[g, Orderless];
g[b, a]
g[a, b]
```

All possible argument orderings are tried to match a definition for an orderless function .

```
ClearAll[g];
SetAttributes[g, Orderless];
g[x_Integer, y_Real] := y^x;
{g[2, 3.0], g[3.0, 2], g[3, 2]}
{9., 9., g[2, 3]}
```

For Flat and Orderless functions, any subset of the arguments may match :

```
ClearAll[h];
SetAttributes[h, {Flat, Orderless}];
(* arguments b,d (of function h) get substituted by x *)
ReplaceAll[ h[a, b, c, d, e] , h[d, b] → x ]
h[a, c, e, x]
```

## (Bubble) Sort and Timing

The algorithm for sorting a list, by repeatedly swapping pairs of successive elements that are found to be out of order, is usually called Bubble Sort .

For a list of Length  $n$ , Bubble Sort will typically take about  $n^2$  steps.

The built-in function **Sort** is faster, as it takes only a little over  $n$  steps.

**Timing**[ expression ] evaluates the expression and returns a list of the time in seconds used, together with the result obtained.

**Timing**[ expression ; ] evaluates the expression and returns a list of the time in seconds used, together with **Null**.

```
SeedRandom[3];
testlistR = RandomSample[Range[10 ^ 2]];

pattern = {x___, b_, a_, y___} /; b > a → {x, a, b, y};
fplR = Timing[
  FixedPointList[
    ReplaceAll[#, pattern] &,
    testlistR
  ];
]
{0.122527, Null}

sortR = Timing[ Sort[testlistR]; ]
{0.000047, Null}
```

---

## More to Explore

Guide to Patterns in *Mathematica*.

[Hyperlink\["https://reference.wolfram.com/language/guide/Patterns.html"\]](https://reference.wolfram.com/language/guide/Patterns.html)  
<https://reference.wolfram.com/language/guide/Patterns.html>

## 21 | Graphs and Networks

A *graph* is a way of showing connections between things—say, how webpages are linked, or how people form a social network.

Let's start with a very simple graph, in which 1 connects to 2, 2 to 3 and 3 to 4. Each of the connections is represented by → (typed as  $\rightarrow$ ).

A very simple graph of connections:

```
Graph[{1 → 2, 2 → 3, 3 → 4}]
```



Automatically label all the “vertices”:

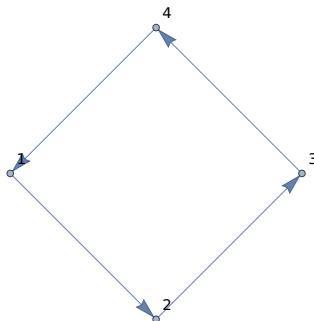
```
Graph[{1 → 2, 2 → 3, 3 → 4}, VertexLabels → All]
```



Let's add one more connection: to connect 4 to 1. Now we have a loop.

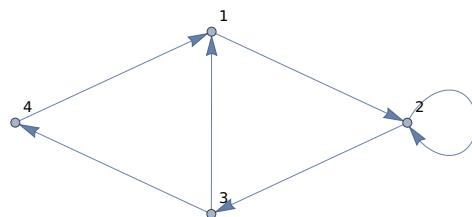
Add another connection, forming a loop:

```
Graph[{1 → 2, 2 → 3, 3 → 4, 4 → 1}, VertexLabels → All]
```



Add two more connections, including one connecting 2 right back to 2:

```
Graph[{1 → 2, 2 → 3, 3 → 4, 4 → 1, 3 → 1, 2 → 2}, VertexLabels → All]
```



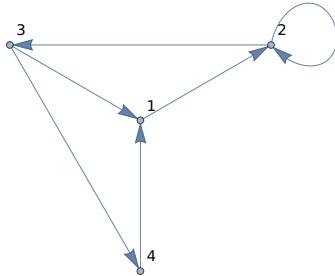
As we add connections, the Wolfram Language chooses to place the vertices or nodes of the graph differently. All that really matters for the meaning,

however, is how the vertices are connected. And if you don't specify otherwise, the Wolfram Language will try to lay the graph out so it's as untangled and easy to understand as possible.

There are options, though, to specify other layouts. Here's an example. It's the same graph as before, with the same connections, but the vertices are laid out differently.

A different layout of the same graph (check by tracing the connections):

```
Graph[{1 → 2, 2 → 3, 3 → 4, 4 → 1, 3 → 1, 2 → 2},
  VertexLabels → All, GraphLayout → "RadialDrawing"]
```



You can do computations on the graph, say finding the shortest path that gets from 4 to 2, always following the arrows.

The shortest path from 4 to 2 on the graph goes through 1:

```
FindShortestPath[Graph[{1 → 2, 2 → 3, 3 → 4, 4 → 1, 3 → 1, 2 → 2}], 4, 2]
{4, 1, 2}
```

Now let's make another graph. This time let's have 3 nodes, and let's have a connection between every one of them.

Start by making an array of all possible connections between 3 objects:

```
Table[i → j, {i, 3}, {j, 3}]
{{1 → 1, 1 → 2, 1 → 3}, {2 → 1, 2 → 2, 2 → 3}, {3 → 1, 3 → 2, 3 → 3}}
```

The result here is a list of lists. But what Graph needs is just a single list of connections. We can get that by using Flatten to “flatten” out the sublists.

Flatten “flattens out” all sublists, wherever they appear:

```
Flatten[{{a, b}, 1, 2, 3, {x, y, {z}}}]
{a, b, 1, 2, 3, x, y, z}
```

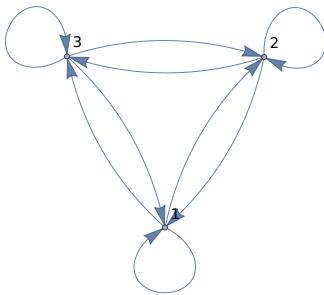
Get a “flattened” list of connections from the array:

```
Flatten[Table[i → j, {i, 3}, {j, 3}]]
```

```
{1 → 1, 1 → 2, 1 → 3, 2 → 1, 2 → 2, 2 → 3, 3 → 1, 3 → 2, 3 → 3}
```

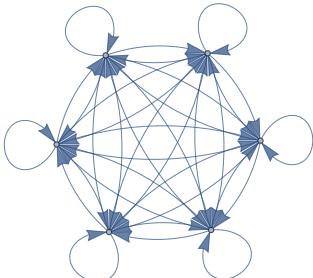
Show the graph of these connections:

```
Graph[Flatten[Table[i → j, {i, 3}, {j, 3}]], VertexLabels → All]
```



Generate the completely connected graph with 6 nodes:

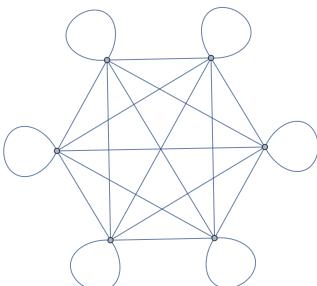
```
Graph[Flatten[Table[i → j, {i, 6}, {j, 6}]]]
```



Sometimes the “direction” of a connection doesn’t matter, so we can drop the arrows.

The “undirected” version of the graph:

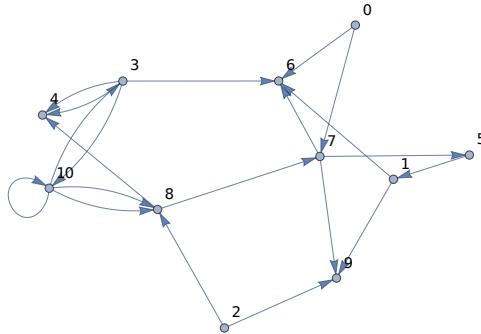
```
UndirectedGraph[Flatten[Table[i → j, {i, 6}, {j, 6}]]]
```



Now let's make a graph with random connections. Here is an example with 20 connections between randomly chosen nodes.

Make a graph with 20 connections between randomly chosen nodes numbered from 0 to 10:

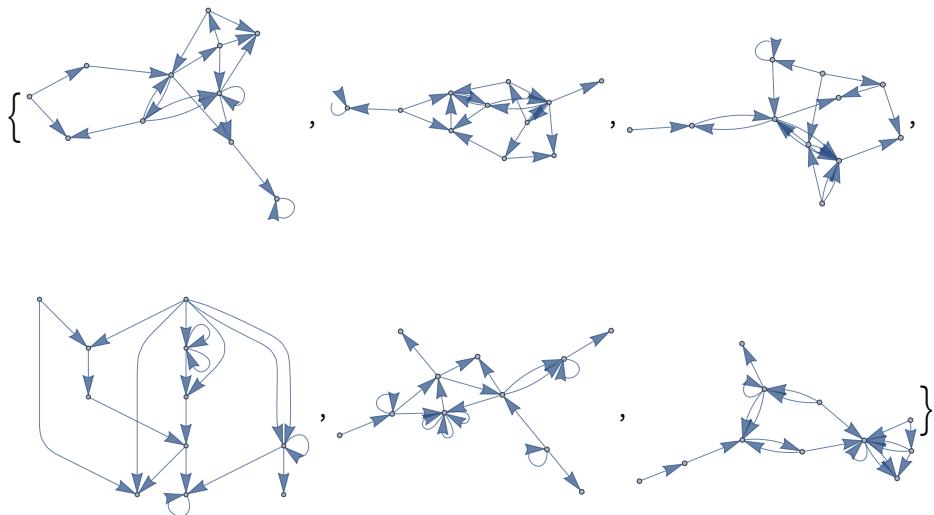
```
Graph[Table[RandomInteger[10] → RandomInteger[10], 20], VertexLabels → All]
```



You'll get a different graph if you generate different random numbers. Here are 6 graphs.

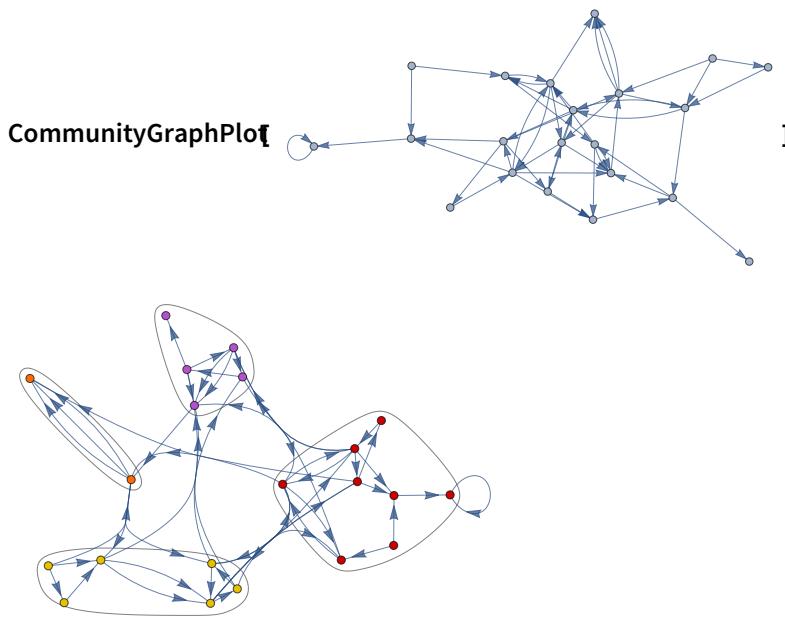
Six randomly generated graphs:

```
Table[Graph[Table[RandomInteger[10] → RandomInteger[10], 20]], 6]
```



There's lots of analysis that can be done on graphs. One example is to break a graph into "communities"—clumps of nodes that are more connected to each other than to the rest of the graph. Let's do that for a random graph.

Make a plot that collects “communities” of nodes together:



The result is a graph with the exact same connections as the original, but with the nodes arranged to illustrate the “community structure” of the graph.

---

### Vocabulary

<b>Graph[{i → j, ...}]</b>	a graph or network of connections
<b>UndirectedGraph[{i → j, ...}]</b>	a graph with no directions to connections
<b>VertexLabels</b>	an option for what vertex labels to include (e.g. All)
<b>FindShortestPath[graph, a, b]</b>	find the shortest path from one node to another
<b>CommunityGraphPlot[list]</b>	display a graph arranged into “communities”
<b>Flatten[list]</b>	flatten out sublists in a list

---

### Exercises

[Preview Exercises](#)

11 Exercises Available  
with 7 extras

[Get Started »](#)

---

## Q&A

### What's the difference between a "graph" and a "network"?

There's no difference. They're just different words for the same thing, though "graph" tends to be more common in math and other formal areas, and "network" more common in more applied areas.

### What are the vertices and edges of a graph?

Vertices are the points, or nodes, of a graph. Edges are the connections. Because graphs have arisen in so many different places, there are quite a few different names used for the same thing.

### How is $i \rightarrow j$ understood?

It's `Rule[i,j]`. Rules are used in lots of places in the Wolfram Language—such as giving settings for options.

### How does the Wolfram Language determine how to lay out graphs?

It uses a variety of methods to try to make them as "untangled" and "balanced" as possible. One method is essentially a mechanical simulation in which each edge is like a spring that tries to get shorter, and every node "electrically repels" others. The option `GraphLayout` lets you specify a layout method to use.

### How big a graph can the Wolfram Language handle?

It's mostly limited by the amount of memory in your computer. Graphs with tens or hundreds of thousands of nodes are not a problem.

### Can I specify annotations to nodes and edges?

Yes. You can give `Graph` lists of nodes and edges that include things like

`Annotation[node,VertexStyle→Red]` or `Annotation[edge, EdgeWeight→20]`. You can extract these annotations using `AnnotationValue`. You can also give overall options to `Graph`. In addition, you can specify "tags" for edges using, for example, `DirectedEdge[i,j,tag]`. Tags are displayed if you use `EdgeLabels→Automatic`.

---

## Tech Notes

- Graphs, like strings, images, graphics, etc., are first-class objects in the Wolfram Language.
- You can enter undirected edges in a graph using `<->`, which displays as  $\longleftrightarrow$ .
- `CompleteGraph[n]` gives the completely connected graph with  $n$  nodes. Among other kinds of special graphs are `GridGraph`, `TorusGraph`, `KaryTree`, etc.
- There are lots of ways to make random graphs (random connections, random numbers of connections, scale-free networks, etc.). `RandomGraph[{100,200}]` makes a random graph with 100 nodes and 200 edges.
- `AdjacencyMatrix[graph]` gives the adjacency matrix for a graph. `AdjacencyGraph[matrix]` constructs a graph from an adjacency matrix.
- Common settings for the `GraphLayout` option are "`SpringElectricalEmbedding`" and "`LayeredDigraphEmbedding`".

- `PlanarGraph[graph]` tries to lay a graph out without any edges crossing—if that's possible.

---

*More to Explore*

[Guide to Graphs and Networks in the Wolfram Language »](#)

## 22 | Machine Learning

So far in this book, when we've wanted the Wolfram Language to do something, we've written code to tell it exactly what to do. But the Wolfram Language is also set up to be able to learn what to do just by looking at examples, using the idea of *machine learning*.

We'll talk about how to train the language yourself. But first let's look at some built-in functions that have already been trained on huge numbers of examples.

`LanguageIdentify` takes pieces of text, and identifies what human language they're in.

Identify the language each phrase is in:

`LanguageIdentify`

```
{ "thank you", "merci", "dar las gracias", "感謝", "благодарить" }]
```

```
{ English, French, Spanish, Chinese, Russian }
```

The Wolfram Language can also do the considerably more difficult “artificial intelligence” task of identifying what an image is of.

Identify what an image is of:

`ImageIdentify`



```
cheetah
```

There's a general function `Classify`, which has been taught various kinds of classification. One example is classifying the “sentiment” of text.

Upbeat text is classified as having positive sentiment:

```
Classify["Sentiment", "I'm so excited to be programming"]
```

Positive

Downbeat text is classified as having negative sentiment:

```
Classify["Sentiment", "math can be really hard"]
```

Negative

You can also train `Classify` yourself. Here's a simple example of classifying handwritten digits as 0 or 1. You give `Classify` a collection of training examples, followed by a particular handwritten digit. Then it'll tell you whether the digit you give is a 0 or 1.

With training examples, `Classify` correctly identifies a handwritten 0:

```
Classify[{0 → 0, 1 → 1, 0 → 0, 1 → 1, 1 → 1, 0 → 0, 0 → 0, 1 → 1, 1 → 1, 0 → 0,
          0 → 0, 0 → 0, 1 → 1, 0 → 0, 1 → 1, 0 → 0, 1 → 1, 1 → 1, 1 → 1},  
0]
```

0

To get some sense of how this works—and because it's useful in its own right—let's talk about the function `Nearest`, that finds what element in a list is nearest to what you supply.

Find what number in the list is nearest to 22:

```
Nearest[{10, 20, 30, 40, 50, 60, 70, 80}, 22]  
{20}
```

Find the nearest three numbers:

```
Nearest[{10, 20, 30, 40, 50, 60, 70, 80}, 22, 3]  
{20, 30, 10}
```

`Nearest` can find nearest colors as well.

Find the 3 colors in the list that are nearest to the color you give:

```
Nearest[{red, orange, yellow, lime, green, cyan, magenta, blue, purple, black}, yellow, 3]  
{yellow, lime, green}
```

It also works on words.

Find the 10 words nearest to "good" in the list of words:

```
Nearest[WordList[], "good", 10]  
{good, food, goad, god, gold, goo, goody, goof, goon, goop}
```

There's a notion of nearness for images too. And though it's far from the whole story, this is effectively part of what `ImageIdentify` is using.

Something that's again related is recognizing text (*optical character recognition* or OCR). Let's make a piece of text that's blurred.

Create an image of the word “hello”, then blur it:

```
Blur[Rasterize[Style["hello", 30]], 3]
```

TextRecognize can still recognize the original text string in this.

Recognize text in the image:

```
TextRecognize[]
```

hello

If the text gets too blurred TextRecognize can’t tell what it says—and you probably can’t either.

Generate a sequence of progressively more blurred pieces of text:

```
Table[Blur[Rasterize["hello"], r], {r, 0, 10}]
```

```
{hello, hello, hello, hello, hello, hello, hello, hello, hello, hello, hello}
```

As the text gets more blurred, TextRecognize makes a mistake, then gives up altogether:

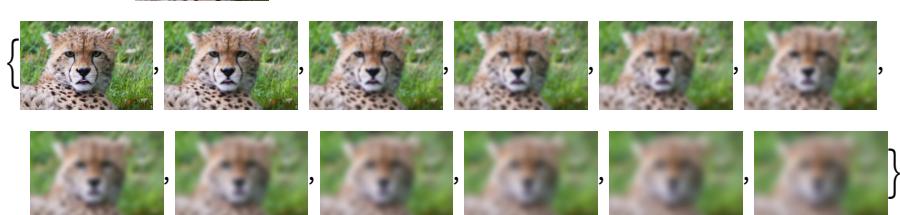
```
Table[TextRecognize[Blur[Rasterize["hello"], r]], {r, 0, 10}]
```

```
{hello, hello, hello, hello, hello, hello, nelle, nelle, ne | le, , }
```

Something similar happens if we progressively blur the picture of a cheetah. When the picture is still fairly sharp, ImageIdentify will correctly identify it as a cheetah. But when it gets too blurred ImageIdentify starts thinking it’s more likely to be a lion, and eventually the best guess is that it’s a picture of a person.

Progressively blur a picture of a cheetah:

```
Table[Blur[, r], {r, 0, 22, 2}]
```



When the picture gets too blurred, `ImageIdentify` no longer thinks it's a cheetah:

```
Table[ImageIdentify[Blur[, r]], {r, 0, 22, 2}]
```

{cheetah, cheetah, cheetah, cheetah, wildcat,  
Canada lynx, feline, nutria, musteline mammal,  
northern river otter, whortleberry, whortleberry}

`ImageIdentify` normally just gives what it thinks is the most likely identification. You can tell it, though, to give a list of possible identifications, starting from the most likely. Here are the top 10 possible identifications, in all categories.

`ImageIdentify` thinks this might be a cheetah, but it's more likely to be a wildcat, or it could be a baboon:

```
ImageIdentify[, All, 10]
```

{wildcat, liger, lynx, Canada lynx, jaguar, mountain lion,  
cheetah, patas monkey, baboon, chacma baboon}

When the image is sufficiently blurred, `ImageIdentify` can have wild ideas about what it might be:

```
ImageIdentify[, All, 10]
```

{whortleberry, blueberry, Alaskan malamute, berry,  
northern fur seal, guadalupe fur seal, California sea lion,  
Australian sea lion, Steller sea lion, sled dog}

In machine learning, one often gives training that explicitly says, for example, “this is a cheetah”, “this is a lion”. But one also often just wants to automatically pick out categories of things without any specific training.

One way to start doing this is to take a collection of things—say colors—and then to find clusters of similar ones. This can be achieved using `FindClusters`.

Collect “clusters” of similar colors into separate lists.

## FindClusters[

```

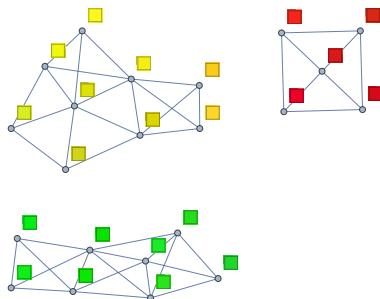
[{"color": "#00FF00"}, {"color": "#FFFF00"}, {"color": "#FF0000"}, {"color": "#008000"}, {"color": "#00FFFF"}, {"color": "#FF00FF"}, {"color": "#800000"}, {"color": "#000080"}, {"color": "#000000"}, {"color": "#0000FF"}, {"color": "#008080"}, {"color": "#808000"}, {"color": "#800080"}, {"color": "#808080"}, {"color": "#000000"}, {"color": "#000000"}, {"color": "#000000"}, {"color": "#000000"}, {"color": "#000000"}, {"color": "#000000"}]

```

You can get a different view by connecting each color to the three most similar colors in the list, then making a graph out of the connections. In the particular example here, there end up being three disconnected subgraphs.

Create a graph of connections based on nearness in “color space”:

```
NearestNeighborGraph[{, , , , , , , , , , , , , , , }, 3, VertexLabels -> All]
```

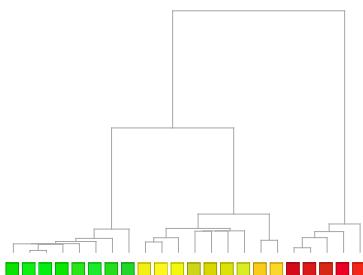


A *dendrogram* is a tree-like plot that lets you see a whole hierarchy of what's near what.

Show nearby colors successively grouped together:

## Dendrogram[

```
{ [ green, yellow, red, green, red, green, red, yellow, yellow, red, yellow, red, green, yellow, green, yellow, orange, green, green ] }
```



When we compare things—whether they’re colors or pictures of animals—we can think of identifying certain *features* that allow us to distinguish them. For colors, a feature might be how light the color is, or how much red it contains. For pictures of animals, a feature might be how furry the animal looks, or how pointy its ears are.

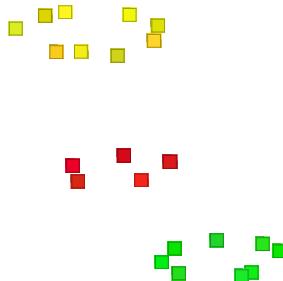
In the Wolfram Language, `FeatureSpacePlot` takes collections of objects and tries to find what it considers the “best” distinguishing features of them, then uses the values of these to position objects in a plot.

FeatureSpacePlot doesn't explicitly say what features it's using—and actually they're usually quite hard to describe. But what happens in the end is that FeatureSpacePlot arranges things so that objects that have similar features are drawn nearby.

FeatureSpacePlot makes similar colors be placed nearby:

## FeatureSpacePlot[

```
{[{"color": "green"}, {"color": "yellow"}, {"color": "red"}, {"color": "green"}, {"color": "green"}, {"color": "red"}, {"color": "green"}, {"color": "yellow"}, {"color": "yellow"}, {"color": "green"}, {"color": "red"}, {"color": "yellow"}, {"color": "red"}, {"color": "green"}, {"color": "green"}, {"color": "yellow"}, {"color": "green"}, {"color": "orange"}, {"color": "green"}, {"color": "green"}]}
```



If one uses, say, 100 colors picked completely at random, then FeatureSpacePlot will again place colors it considers similar nearby.

100 random colors laid out by FeatureSpacePlot:

## FeatureSpacePlot[RandomColor[100]]



Let's try the same kind of thing with images of letters.

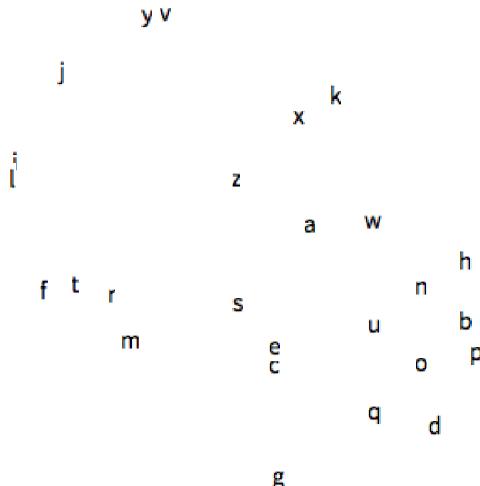
Make a rasterized image of each letter in the alphabet:

```
Table[Rasterize[FromLetterNumber[n]], {n, 26}]
```

{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}

FeatureSpacePlot will use visual features of these images to lay them out. The result is that letters that look similar—like y and v or e and c—will wind up nearby.

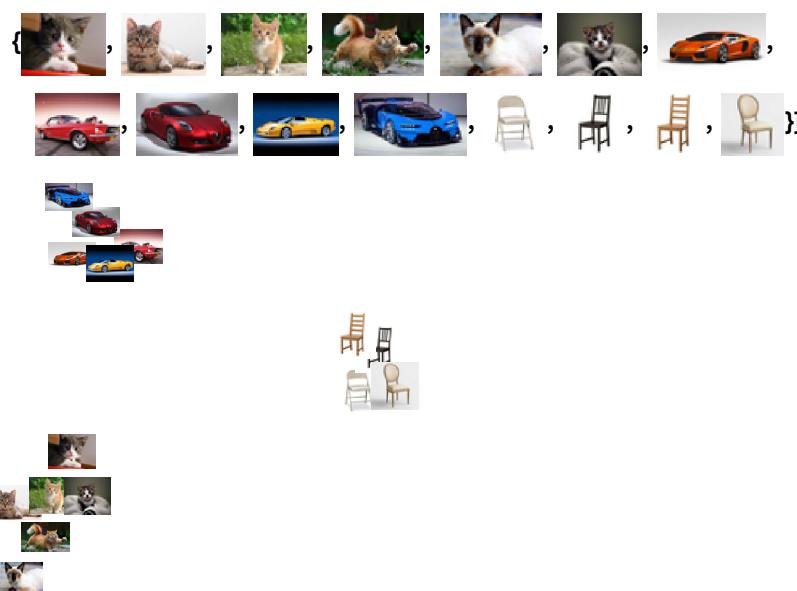
```
FeatureSpacePlot[Table[Rasterize[FromLetterNumber[n]], {n, 26}]]
```



Here's the same thing, but now with pictures of cats, cars and chairs. FeatureSpacePlot immediately separates the different kinds of things.

FeatureSpacePlot places photographs of different kinds of things quite far apart:

```
FeatureSpacePlot[
```



## Vocabulary

<b>LanguageIdentify[text]</b>	identify what human language text is in
<b>ImageIdentify[image]</b>	identify what an image is of
<b>TextRecognize[text]</b>	recognize text from an image (OCR)
<b>Classify[training, data]</b>	classify data on the basis of training examples
<b>Nearest[list, item]</b>	find what element of list is nearest to item
<b>FindClusters[list]</b>	find clusters of similar items
<b>NearestNeighborGraph[list, n]</b>	connect elements of list to their n nearest neighbors
<b>Dendrogram[list]</b>	make a hierarchical tree of relations between items
<b>FeatureSpacePlot[list]</b>	plot elements of list in an inferred “feature space”

## Exercises

+ Preview Exercises

16 Exercises Available  
with 7 extras

Get Started »

## Q&A

### How come I'm getting different results from the ones shown here?

Probably because Wolfram Language machine learning functions are continually getting more training—and so their results may change, hopefully always getting better. For `TextRecognize`, the results can depend in detail on the fonts used, and exactly how they're rendered and rasterized on your computer.

### How does `ImageIdentify` work inside?

It's based on artificial neural networks inspired by the way brains seem to work. It's been trained with millions of example images, from which it's progressively learned to make distinctions. And a bit like in the game of “twenty questions”, by using enough of these distinctions it can eventually determine what an image is of.

### How many kinds of things can `ImageIdentify` recognize?

At least 10,000—which is more than a typical human. (There are about 5000 “picturable nouns” in English.)

### What makes `ImageIdentify` give a wrong answer?

A common cause is that what it's asked about isn't close enough to anything it's been trained on. This can happen if something is in an unusual configuration or environment (for example, if a boat is not on a bluish background). `ImageIdentify` usually tries to find some kind of match, and the mistakes it makes often seem

very “humanlike”.

### Can I ask ImageIdentify the probabilities it assigns to different identifications?

Yes. For example, to find the probabilities for the top 10 identifications in all categories use `ImageIdentify[image, All, 10, "Probability"]`.

### How many examples does Classify typically need to work well?

If the general area (like everyday images) is one it already knows well, then as few as a hundred. But in areas that are new, it can take many millions of examples to achieve good results.

### How does Nearest figure out a distance between colors?

It uses the function `ColorDistance`, which is based on a model of human color vision.

### How does Nearest determine nearby words?

By looking at those at the smallest `EditDistance`, that is, reached by the smallest number of single-letter insertions, deletions and substitutions.

### What features does FeatureSpacePlot use?

There's no easy answer. When it's given a collection of things, it'll learn features that distinguish them—though it's typically primed by having seen many other things of the same general type (like images).

---

## Tech Notes

- The Wolfram Language stores its latest machine learning classifiers in the cloud—but if you're using a desktop system, they'll automatically be downloaded, and then they'll run locally.
- `BarcodeImage` and `BarcodeRecognize` work with bar codes and QR codes instead of pure text.
- `ImageIdentify` is the core of what the [imageidentify.com](http://imageidentify.com) website does.
- If you just give `Classify` training examples, it'll produce a `ClassifierFunction` that can later be applied to many different pieces of data. This is pretty much always how `Classify` is used in practice.
- You can get a large standard training set of handwritten digits using `ResourceData["MNIST"]`.
- `Classify` automatically picks between methods such as *logistic regression*, *naive Bayes*, *random forests* and *support vector machines*, as well as *neural networks*.
- `FindClusters` does *unsupervised machine learning*, where the computer just looks at data without being told anything about it. `Classify` does *supervised machine learning*, being given a set of training examples.
- `Dendrogram` does *hierarchical clustering*, and can be used to reconstruct evolutionary trees in areas like bioinformatics and historical linguistics.
- `FeatureSpacePlot` does *dimension reduction*, taking data that's represented by many parameters, and finding a good way to “project” these down so they can be plotted in 2D.
- `Rasterize/@Alphabet[]` is a better way to make a list of rasterized letters, but we won't talk about `/@` until Section 25.
- `FeatureExtraction` lets you get out the feature vectors used by `FeatureSpacePlot`.

- FeatureNearest is like Nearest except that it learns what should be considered near by looking at the actual data you give. It's what you need to do something like build an image-search function.
  - NetModel gives you immediate access to a large number of prebuilt nets from the continually updated Wolfram Neural Net Repository.
  - You can build and train your own neural nets in the Wolfram Language using functions like NetChain, NetGraph and NetTrain. You can include prebuilt nets from NetModel.
- 

### *More to Explore*

[Guide to Machine Learning in the Wolfram Language »](#)