2. Elementi fondamentali del linguaggio

▪ Riscrittura di termini

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

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

Vediamone un esempio:

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

$4\, a^2$

Questi sono i passi seguiti.
Nella espressione :

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

il Kernel riscrive a*a come a^2

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

Dopodiché riscrive D[a^3, a] come 3 a^2

```
a ^ 2 + 3 a ^ 2;
```

Infine riconosce che a^2+3 a^2 può essere riscritto come 4 a^2

```
4 a ^ 2;
```

Vediamolo con Trace

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

$\{\{a\, a,\ a^2\},\ \{\partial_a a^3,\ 3\, a^2\},\ a^2 + 3\, a^2,\ 4\, a^2\}$

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

⌘ 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 più grandi sono composte da espressioni più piccole, che a loro volta sono composte da espressioni via via più piccole, fino ad arrivare agli **atomi** (sotto-espressioni che non possono essere suddivise) del linguaggio.

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

2. Elementi fondamentali del linguaggio

- **Riscrittura di termini**

2.1 Espressioni: espressioni normali (2.1.1)

⌘ Ogni cosa in *Mathematica* è una espressione.

Esistono fondamentalmente **due** tipi di espressione: atomica e normale.
Gli **atomi** possono essere simboli, numeri o stringhe di caratteri (cfr. 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]];
```

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

A sua volta:

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

è una espressione normale:
- la sua Head è un atomo (il simbolo Log);
- la sue **due** parti sono il numero reale 2.5 ed il numero intero 7.

⌘ La sintassi delle espressioni è disegnata per essere simile al costrutto di *chiamata di funzione* in linguaggi 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.

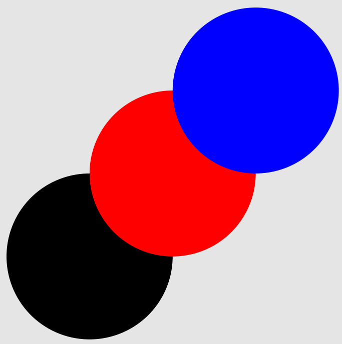
⌘ Non ogni espressione normale, tuttavia, può essere pensata come una chiamata di funzione.
Una espressione, infatti, può semplicemente rappresentare dati.
Ad esempio:

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

è una direttiva grafica: essa dice al Kernel che una data primitiva grafica (cui RGBColor[1,0,0] è associata) deve essere resa in colore rosso.
Non vi è alcuna chiamata di funzione associata al simbolo RGBColor e l'espressione RGBColor[1,0,0] non può essere riscritta in alcun modo.

```
? RGBColor  (* Built-in Symbol *)
```

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



⌘ Ogni espressione in *Mathematica* può essere costruita usando solo **tre** blocchi di costruzione sintattica: atomi, virgole, 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 (usando atomi, virgole e parentesi quadre):

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

⌘Il parser di *Mathematica* converte input, quali a*a in Power[a, 2], a^3 in Power[a,3] e così via.

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

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

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

```
(* Map *)
mylist = {x * y , a + a , a + b};
mylist // FullForm
(* Map  applica  FullForm  agli elementi  della  lista  mylist *)
Map[ FullForm , mylist]
```

```
List[Times[x, y], Times[2, a], Plus[a, b]]
```

```
{Times[x, y], Times[2, a], Plus[a, b]}
```

⌘ 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 pure *forme speciali di output*.
È 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. 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,* è una espressione che non può essere suddivisa in espressioni più piccole.

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

- **Simboli**

Un simbolo è 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 più potenti, dato che non è 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 è meramente un sostituto (proxy) per un dato.

```
(* Esempio di calcolo simbolico.
   Il risultato è 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
```

```
14.0.0 for Linux x86 (64-bit) (December 13, 2023)
```

```
$Version = 3
```

Set: Symbol $Version is Protected.

```
3
```

- **Numeri**

- **Stringhe di caratteri**

Una stringa di caratteri (o semplicemente una stringa) è una qualsiasi sequenza di caratteri, racchiusa tra una coppia di doppi apici.

```
"Hello world"
(* Questa è una stringa.
   N.B. Mathematica non stampa la coppia di apici, quando stampa la stringa *)
```

```
Hello world
```

Possiamo usare InputForm per vedere che l'output è effettivamente una stringa

```
"Hello world";
InputForm[%]
(* InputForm[expr] stampa una versione di
 expr adatta ad essere un input per Mathematica *)
```

```
"Hello world"
```

In una stringa, la sequenza di caratteri \" sta per il (singolo) doppio apice "
Di conseguenza, l'input seguente è pure una stringa:

```
"For example, \"Hello world\" is a string."
```

```
For example, "Hello world" is a string.
```

Ci sono molte Built-in per agire sulle stringhe di caratteri, quali quelle per determinare la loro lunghezza, concatenarle, farne uno shift da maiuscola a minuscola (e viceversa), individuare e rimpiazzare sottostringhe (cfr. EIWL 11 e/o 3.6 "Stringhe di caratteri") e cosi' via.

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,* è una espressione che non può essere suddivisa in espressioni più 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 *dddddd*

Rational: ha la forma *intero1/intero2*

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

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

```
(* Integer *)
1 234 567 890
```

```
1 234 567 890
```

```
(* Rational *)
2 / 3
```

$$\frac{2}{3}$$

```
(* Real *)
nr = 21345.6789;
(* Di default , in  output vengono mostrate 6  cifre  significative *)
(* NumberForm può 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 back-tick nell'output di FullForm[nr] è un NumberMarks,
ossia un argomento opzionale che può essere usato per
 specificare quante cifre significative devono essere stampate *)
```

```
21345.7
21345.6789
21345.6789`
MachinePrecision
```
2.13457×10^4

2.13456789×10^4

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

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

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

```
(* Rationalize *)
nr = 21 345.6789;
Rationalize[nr]
Rationalize[nr, 10^-7]
(* Rationalize[x] restituisce p/q se Abs[p/q-x]<1/(100 q)^2 *)
(* Se Rationalize[x] restituisce x,
significa che nessun numero razionale soddisfa la disequazione qui sopra *)
(* In questo caso, proviamo a specificare una tolleranza *)
```

```
21 345.7
```

$$\frac{213\,456\,789}{10\,000}$$

```
(* N  numericizza *)
nint = 123 456 789 * 10^-4;
tt = {N[nint],  N[nint, 20]}
Map[FullForm, tt]
(* N[nint,20]  è  un BigNumber, ossia un numero a precisione arbitraria:
  qui ha precisione 20  *)
Map[MachineNumberQ, tt]
```

```
{12 345.7, 12 345.678900000000000}
```

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

```
{True, False}
```

```
(* Complex *)
(* Notare il terzo l'output di cc :
  MachinePrecision si propaga come un virus *)
cc = {2/3 + 4 I,
  2/3 + (45/10) I,
  2/3 + 4.5 I}
```

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

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

⌘ L'esempio che segue è 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 è un intero di 52 cifre *)
(*  β = base , η = esponente *)
{{β, η} = {5 , 73};
 out = β^η ,
 Map[Precision, {β, η, out }]}
```

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

```
(* REALE:  Un input in precisione macchina → Output in precisione macchina *)
{{β, η} = {5. , 73};
 out2 = β^η,
 Map[Precision, {β, η, out2 }]}
(* la Machine Precision  è  virale *)
{{β, η} = {5 , 73.};
 out3 = β^η,
 Map[Precision, {β, η, out3 }]}
```

```
{1.05879 × 10^51, {MachinePrecision, ∞, MachinePrecision}}
```

```
{1.05879 × 10^51, {∞, MachinePrecision, MachinePrecision}}
```

```
(* REALE (precisione arbitraria , Big Number):
 Qui, l' input a precisione più bassa ha (circa) 25 cifre decimali.
  Output ha (circa) 23 cifre decimali *)
η = 73;
(* β ha 24 zeri *)
β = 5.000000000000000000000000;
{out4 = β^η, Map[Precision, {β, out4}]}
```

```
{1.0587911840678754238354 × 10^51, {24.699, 22.8356}}
```

⌘ La presenza del punto decimale, nell'input, è interpretata come voler significare che l'input è approssi mato (non esatto) ed è 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 è partiti dalla precisione di circa 25 e si è arrivati ad una precisione di circa 23):

```
(* Precision fornisce il numero di
 cifre significative in un numero approssimato. *)
β = 5.000000000000000000000000;
η = 73;
out4 = β ^ η;
{Precision[β], Precision[out4]}
(* NB. Il risultato di Precision può non corrispondere
 esattamente al numero di cifre mostrato nella cella.
  Il motivo è che la precisione numerica viene
 calcolata in base binaria e poi convertita in decimale.  *)
```

```
{24.699, 22.8356}
```

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

```
η = 73;
(* Input con 25 cifre decimali . Output, qui, ha (circa) 24 cifre decimali *)
β = 5`25 ;
{out5 = β ^ η, Map[Precision, {β, out5 }]}

β = N[5, 25] ;
{ out6 = β ^ η, Map[Precision, {β, out6 }]}
```

{1.05879118406787542383540 × 10^{51}, {25., 23.1367}}

{1.05879118406787542383540 × 10^{51}, {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 può variare su differenti architetture.

⌘ La funzione MachineNumberQ[ ] può essere usata per determinare se un numero approssimato è un numero macchina

```
MachineNumberQ[5.000000000000000000000000]
(* 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 è svolto in hardware a virgola mobile *)
```

1.05879×10^{51}

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

⌘ Per vedere tutte le cifre, possiamo usare FullForm[ ] .
FullForm[expr] stampa la forma interna di qualsiasi cosa presente nell’espressione expr

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

$\{1.05879 \times 10^{51}$, 1.0587911840678756`*^51$\}$

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

$\sqrt{2} + a$

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

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

```
1.41421 + a
```

```
Plus[1.4142135623730951`, a]
```

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

⌘ In alternativa, si può usare InputForm[ ].
InputForm[ ] fa vedere (displays) ciò che si deve scrivere in input per ottenere qualcosa di uguale alla espressione data

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

```
1.0587911840678756*^51
```

```
5.^73 == 1.0587911840678756*^51
```

```
True
```

```
(* InputForm[ Sqrt[2]+a] *)
```

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

```
1.4142135623730951 + a
```

```
Sqrt[2.] + a == 1.4142135623730951 + a
```

```
True
```

```
? InputForm
```

Nota.
La precisione di un numero macchina è sempre $MachinePrecision.
Al contrario, se la precisione di un numero arbitrario è uguale od inferiore a $MachinePrecision, tale numero viene 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. $MachinePrecision è il valore numerico del simbolo MachinePrecision *)
{FullForm[$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 può 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 è sempre considerato essere un numero approssimato, anche quando noi sappiamo che non lo è.
D'altra parte *Mathematica* non può fare ipotesi diverse, altrimenti si incorrerebbe in errori:

```
(* Il risultato di questo calcolo è 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
```

```
∞
```

Per riassumere.
Ci sono due classi 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 Sqrt[3] come un numero, perché per fare ciò dovrebbe inserire una approssimazione (dato che Sqrt[3] non ha una rappresentazione decimale finita).

⌘ D'altra parte, *Mathematic*a valuta invece:

```
Sqrt[3.]
```

```
1.73205
```

L'argomento **3.** (della funzione Sqrt[ ]) è considerato approssimato, perché contiene la virgola mobile.
Dato che il numero **3.** è già 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 funzione N (come già visto):

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

```
1.73205
```

```
MachinePrecision
```

```
{1.7320508075688772`, 1.7320508075688772}
```

⌘ 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 uno degli argomenti (passati ad una funzione built-in numerica) è approssimato, la funzione verrà valutata, convertendo tale argomento a numero approssimato, nella precisione più alta giustificabile.
Rivediamolo con un esempio:

```
(*  Qui, l'addendo  è un numero macchina *)
add = 2.5;
Precision[add]
(* Il risultato è un numero macchina *)
{sum = 1 + add,
 Precision[sum]}
```

```
MachinePrecision
```

```
{3.5, MachinePrecision}
```

```
(* Qui, l'addendo ha 31 cifre dopo la virgola  *)
addendo = 2.5000000000000000000000000000001;
Precision[addendo]
(* Il risultato è un numero in precisione arbitraria *)
{sum = 1 + addendo,
 Precision[sum]}
```

```
31.3979
```

```
{3.5000000000000000000000000000001, 31.5441}
```

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

```
{0, 0.}
```

```
(* Un altro esempio di perdita di precisione nell'output *)
addendo2 = 1.0000000090000000000000000000001;
Precision[addendo2]
(* Abbiamo perso 8 cifre con una sola sottrazione *)
{subtract = 1 - addendo2, Precision[subtract]}
```

```
31.
```

$\{-9.0000000000000000000000 \times 10^{-9}, 22.9542\}$

⌘ Un buon riferimento è 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. 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	**addition**	Plus
5 - 2 == 5 + (-2)	**subtraction**	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

1.1 Compute 1 + 2 + 3 (Plus, Sum) ⟹ 6

1.2 Add the numbers 1, 2, 3, 4, 5 ⟹ 15

1.3 Multiply the numbers 1, 2, 3, 4, 5 (Times, Factorial) ⟹ 120

1.4 Compute 5 squared (i.e. 5*5 or 5 raised to the power 2: Power)⟹25

1.5 Compute 3 raised to the fourth power ⟹ 81

1.6 Compute 10 raised to the power 12 ⟹ 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 ⟹2117000

+1.1 Add all integers from - 3 to + 3 (Sum) ⟹ 0

+1.2 Compute 24 divided by 3 ⟹ 8 (Division, Power, Reciprocal)

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

+1.4 Subtract 5 squared from 100 ⟹ 75 (Various ways to get 75 from 5 and 10)

+1.5 Multiply 6 by 5 squared, and add 7 ⟹157 (Operation priorities, Trace)

+1.6 Compute 3 squared minus 2 cubed (Trace) ⟹1

+1.7 Compute 2 cubed times 3 squared (Trace)⟹ 72

+1.8 Compute "double the sum of 8 and negative 11" ⟹ -6

Q & A

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

```
{24 / 3, N[24 / 3], 24 / 3., 24. / 3, N[24 / 3, 30]}
{Precision[24 / 3], Precision[N[25 / 3]], Precision[N[25 / 3, 30]]}
```

```
{8, 8., 8., 8., 8.00000000000000000000000000000}
```

```
{∞, MachinePrecision, 30.}
```

```
$MachinePrecision
```

```
15.9546
```

```
{24 / 5, FullForm[24 / 5], N[24 / 5],  N[24 / 5, 30]}
{Precision[24 / 5], Precision[N[25 / 5]], Precision[N[25 / 5, 30]]}
```

$\left\{\frac{24}{5}, \text{Rational}[24, 5], 4.8, 4.80000000000000000000000000000\right\}$

```
{∞, MachinePrecision, 30.}
```

What happens if I compute 1/0?

```
1 / 0
```

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

```
ComplexInfinity
```

Introducing Built-in 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 + 4 using **Plus** :

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

```
7
```

Times does multiplication :

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

You can put functions inside other functions:

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

```
10
```

```
(* Replace[]  /. *)
```

▫ All functions use **square** brackets for their arguments.

▫ A Built-in function name starts with a **Capital** letter.
If the name is compound, each word starts with a Capital letter.

```
? LinearSolve
```

Symbol

LinearSolve[m, b] finds an x that solves the matrix equation $m.x == b$.
LinearSolve[m] generates a
LinearSolveFunction[...] that can be applied repeatedly to different b.

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

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

```
7
```

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

```
? Random
```

Symbol

Random[] gives a uniformly distributed pseudorandom Real in the range 0 to 1.
Random[*type*, *range*] gives a pseudorandom number of the
specified type, lying in the specified range. Possible types are: Integer,
Real and Complex. The default range is 0 to 1. You can give the range
{*min*, *max*} explicitly; a range specification of *max* is equivalent to {0, *max*}.

Documentation Local » | Web »
Attributes {Protected}
Full Name System`Random

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

```
4
```

```
64
```

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

```
61
```

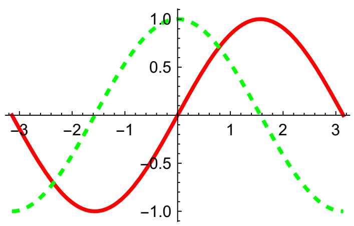
```
(* Organising inputs *)
{test = 4 , Set[test, 4],
 6 × 3 , 6 * 3, Times[6, 3]}
```

```
{4, 4, 18, 18, 18}
```

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

Built-in di grafica

```
(* Esempio di Plot : si veda l' Help di Plot *)
Plot[
 {Sin[x], Cos[x]},
 {x, -Pi, Pi},
 (*  →   Rule[  , ]  *)
 PlotStyle → {Red, {Dashed, Green}},
 ImageSize → Small
]
```



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

Replace

FullForm

tutorial/BasicObjects

Exercises

2.1 Compute 7+6+5 using Plus ⟹ 18

2.2 Compute 2×(3+4) using Times and Plus ⟹ 14

2.3 Use Max to find the larger of 6×8 and 5×9 ⟹ 48

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

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

+2.1 Compute 5×4×3×2 using Times (Plus[] is 0; Times[] is 1) ⟹120

+2.2 Compute 2–3 using Subtract ⟹ -1

+2.3 Compute (8+7)*(9+2) using Times and Plus ⟹165

+2.4 Compute (26–89)/9 using Subtract and Divide ⟹ -7

+2.5 Compute 100–5^2 using Subtract and Power ⟹75

+2.6 Find the larger of 3^5 and 5^3 (Print) ⟹ 5^3

+2.7 Multiply 3 and the larger of 4^3 and 3^4 (Max) ⟹3×3^4

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

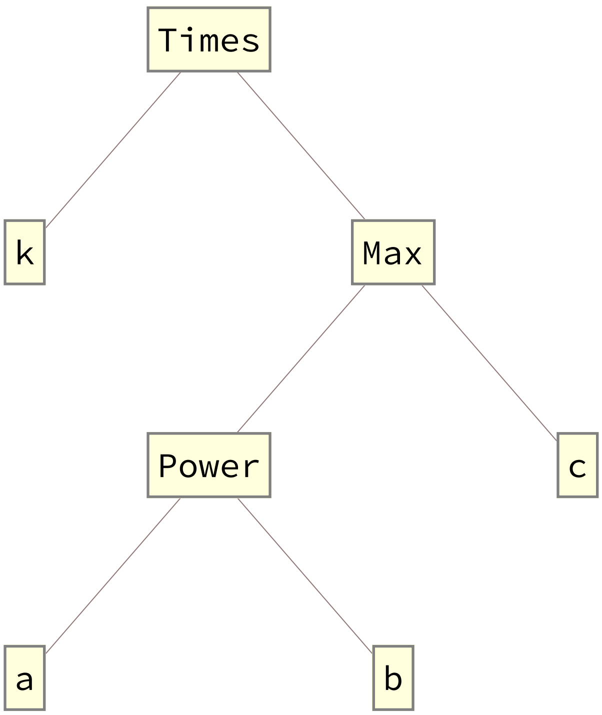
Tech Notes

```
(* 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. 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 è 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 ciò 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[ ]
È 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.

```
Trace[Sin[Log[2.5, 7]]]
(* Come vedremo tra poco, FullForm[Sin[Log[2.5,7]]] e TreeForm[Sin[Log[2.5,7]]]
 restituiscono subito 0.851012661490406`,che è un numero a precisione macchina *)
```

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

```
(* Ricordo che Log[a,b]  viene riscritto  come  Log[b]/Log[a]*)
 Log[a, b] == Log[E, b] / Log[E, a] ==  Log[b] / Log[a]
Trace[ Log[a, b]];
```

```
True
```

Nota. Per esteso, i passi della valutazione di Sin[Log[2.5,7]] sono :
(1a) Log[2.5, 7] viene riscritto come Log[7]/Log[2.5]

(1b) Log[7] viene valutato numericamente

(1c) Log[2.5] viene valutato numericamente

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

2. 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 *) Log[2.5, 7] == Log[7] / Log[2.5]
(* 1b *) {Log[7] , Log[7.]}
(* 1c *) Log[2.5]
(* 1 *) Log[7.] / Log[2.5]
(* 2 *) Sin[%]
```

```
True
```

```
{Log[7], 1.94591}
```

```
0.916291
```

```
2.12368
```

```
0.851013
```

Valutazione standard (e non-standard)

⌘ L'esempio appena visto mosta un punto importante dell processo di valutazione.
In generale, le parti di una espressione normale vengono valutate prima dell'intera espressione.
Questo modo di procedere è 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 è detto *valutazione non-standard* .
Ne vedremo un esempio più avanti.

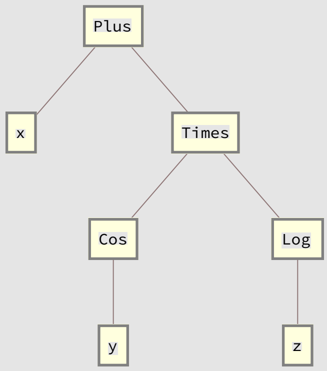
⌘ In termini informatici, una espressione è un albero (tree) e la sua valutazione viene eseguita "depth-first" i.e. vengono valutate per prime le parti (sotto-espressioni) che stanno a maggiore profondità (foglie) nell'albero rappresentante l'espressione stessa.

⌘ Le parentesi graffe indicano il livello di profondità (Depth) della sotto-espressione correntemente valutata.
L'annidamento delle parentesi graffe diventa maggiore via via che il processo di valutazione si addentra nella 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[ ] può risultare non troppo leggibile, specie per espressioni molto grandi ed articolate.
In tale caso, è meglio usare FullForm[ ], studiandola attentamente.

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

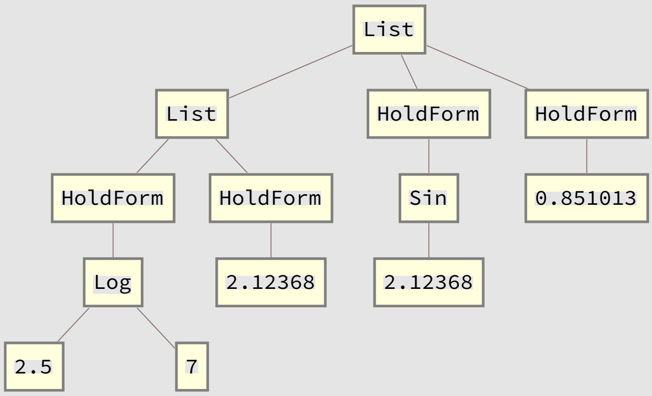


HoldForm[ ]

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 = Trace[Sin[Log[2.5, 7]]]
trace // TreeForm
```

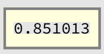
```
{{Log[2.5, 7], 2.12368}, Sin[2.12368], 0.851013}
```



⌘ 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,
che risulta dalla valutazione complessiva *)
espressione = Sin[Log[2.5, 7]];
TreeForm[espressione, ImageSize → Small]
FullForm[espressione]
```




```
0.851012661490406`
```

⌘ Per vedere la struttura interna di una espressione, la cui valutazione complessiva restituisce un numero (e.g. Sin[Log[2.5, 7]]), è necessario inibirne la valutazione.

Per fare ciò, possiamo impacchettare tale espressione dentro una head che impedisca ai suoi argomenti di essere valutati .
Un esempio di tale head è, appunto, HoldForm[ ] .
⌘ HoldForm[ ] inibisce la valutazione dei suoi argomenti. In questo modo, permette di vedere esplicitata la struttura di una espressione (cui HoldForm[ ] sia stata applicata).
⌘ HoldForm[ ] realizza, pertanto, una valutazione non-standard

```
? HoldForm
```

Symbol

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

⌘ Un buon riferimento è 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[] è inibita la valutazione *)
(* HoldForm ha la caratteristica Protected
 i.e. il suo nome non è 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 più 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 è, appunto, un componente di una Trace[ ]).

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, però, in cui l'affermazione precedente sembra essere falsa.
Un esempio è dato dall'operatore SetDelayed[ ] (cfr. 2.3.4), che pare non restituire alcun valore:

```
s2 = Sqrt[2] (* Set *)
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
```

Symbol

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 *)
(* Il punto-e-virgola inibisce l'output di Set *)
```

⌘ Null appare se è parte di una espressione più grande:

```
s2 = Sqrt[2];
1 + %
```

$1 + \sqrt{2}$

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

```
1 + Null
```

```
Clear[a, b];

SeedRandom[3];
a = RandomReal[];
{a, a, a}

SeedRandom[3];
b := RandomReal[];
{b, b, b}
```

```
{0.478554, 0.478554, 0.478554}
```

```
{0.478554, 0.00869692, 0.347029}
```

⌘ 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 è interpretato come prodotto *) ×
     tmp
    (* Pertanto, viene restituito Null
     (esito di While[]) moltiplicato per il valore salvato in tmp *)
  ];
f2[100.]
```

```
1.77828
```

```
1.77828 Null
```

⌘ Null appare se sopprimiamo esplicitamente un output (magari perche' è troppo grande, oppure perché 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]]; ]
```

```
{0.000118, 7 620 753 696}
```

```
{0.000091, Null}
```

⌘ SetDelayed[ ] è un esempio di funzione che opera producendo un *effetto collaterale* (side effect):
il risultato atteso dalla esecuzione della funzione non è il *return value*, ma è 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* è la creazione di una **regola** di riscrittura per il simbolo s3

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

```
3
```

⌘ DISGRESSIONE (su TypeOf e Return)

Attenzione alle dipendenze cicliche

⌘ Per completare questa sezione, dobbiamo analizzare un ultimo argomento.
Una assunzione implicita, nel processo di valutazione, è 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 può essere riscritto come yang *)
yang := yin
(* il secondo statement dice al Kernel che yang può essere riscritto come yin *)
```

⌘ Per superare un caso come quello qui sopra, nel Kernel (per fortuna) è 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 è avvenuta per 4096 (2^12) 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) è una circostanza fortuita, dovuta alle definizioni usate.

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

⌘ Message[ ]

La funzione Message[ ] causa l'apparizione del messaggio di errore (in rosso) sull'avere superato $IterationLimit .

⌘ La funzione Message[ ] viene invocata direttamente dal Kernel e non è parte della espressione originale (e neppure di qualsiasi delle sue forme intermedie).
Message[ ] può 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, ...] prints a message, inserting the values of the e_i as needed.

Esercizio 1 pg. 30

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

Perchè, 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 può, successivamente, ampliare a 90 la sua precisione.

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

```
1.7320508075688772`
```

```
MachinePrecision
```

Un modo corretto per ottenere Sqrt[3] con 90 cifre di precisione è il seguente (cfr. 2.1.2):

```
(* exact3 = Sqrt[3];
tre=N[ exact3 , 90]; *)
tre90 = N[Sqrt[3], 90]
tre90 // Precision
```

```
1.73205080756887729352744634150587236694280525381038062805580697945193301690880003\
 708114619
```

```
90.
```

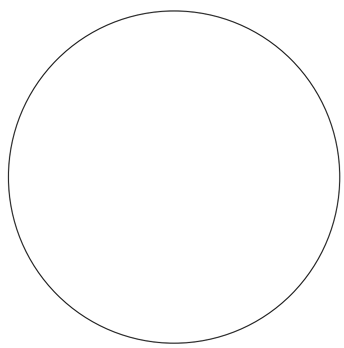
First Look at Lists

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, "b", Graphics[Circle[]]}
```

$\left\{1,\ \frac{1}{2},\ 0.666667,\ \pi,\ a,\ b,\right.$  $\left.\right\}$

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

```
nestedList = {{1, 2}, {4, 5, 6}}; TableForm[nestedList]
```

```
1   2
4   5   6
```

```
matrix = {{1, 2, 3}, {4, 5, 6}}; MatrixForm[matrix]
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

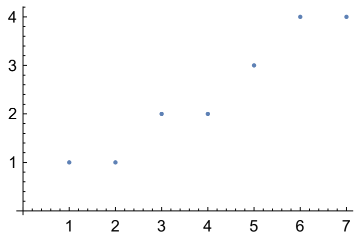
```
(* Insiemistica, Set Theory *)
? Intersection
```

Symbol

Intersection[$list_1$, $list_2$, …] gives a sorted list of the elements common to all the $list_i$.

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.

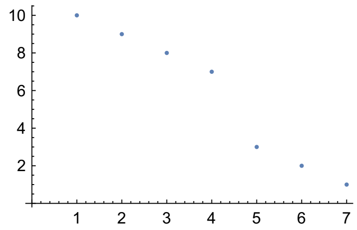
```
(* myList={1,1,2,2,3,4,4}; *)
ListPlot[myList] == ListPlot[{{1, 1}, {2, 1}, {3, 2}, {4, 2}, {5, 3}, {6, 4}, {7, 4}}]
```

```
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}}]
```

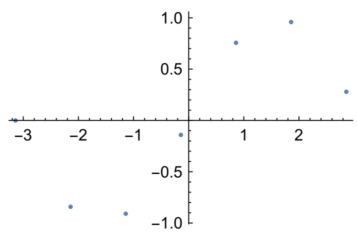
```
True
```

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



```
(* A table of points in the plane,
whose coordinates are the numerical values
  {t, Sin[t]} in [-π, π] ; *)
tt = Table[{N[t], N[Sin[t]]}, {t, -Pi, Pi}];
TableForm[tt]
ListPlot[tt, ImageSize → Small]
```

-3.14159	0.
-2.14159	-0.841471
-1.14159	-0.909297
-0.141593	-0.14112
0.858407	0.756802
1.85841	0.958924
2.85841	0.279415

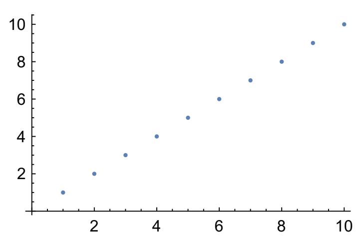


DIGRESSION (Interpolation, Show, Graphics)

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}



```
? Range
```

Symbol

Range[i_{max}] generates the list {1, 2, …, i_{max}}.

Range[i_{min}, i_{max}] generates the list {i_{min}, …, i_{max}}.

Range[i_{min}, i_{max}, di] uses step di.

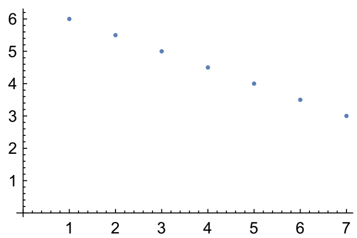
Documentation Local » | Web »

Attributes {Listable, Protected}

Full Name System`Range

```
myRange = Range[6, 3, -1 / 2]
ListPlot[myRange, ImageSize → Small]
```

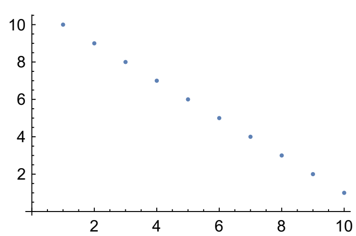
$\left\{6, \frac{11}{2}, 5, \frac{9}{2}, 4, \frac{7}{2}, 3\right\}$



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

```
myRange = Range[10];
reversed = Reverse[myRange]
ListPlot[reversed, ImageSize → Small]
```

{10, 9, 8, 7, 6, 5, 4, 3, 2, 1}

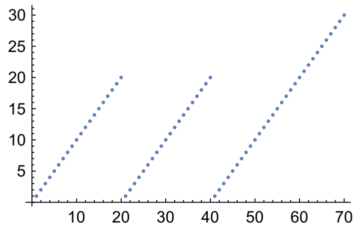


Join joins lists together and creates a single **List** .

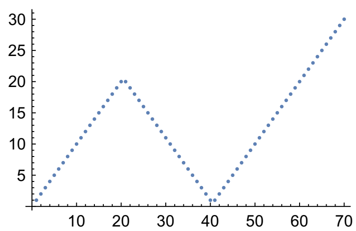
```
(* Elementes are unsorted *)
Join[{4, 5}, {1, 2, 3}, {6, 7}]
(* Duplicates are kept *)
Join[{1, 2, 3}, {1, 2, 3, 4, 5}]
```

{4, 5, 1, 2, 3, 6, 7}

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

(*  Union : duplicates eliminated and elements 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	
Graphics	
GraphicsRow	
Import, Export	

$ImportFormats, $ExportFormats
Set (=) , SetDelayed (:=)
Equal (==) , SameQ (===)
Function (pure function)
Rule (→), RuleDelayed (:→)
Clear
Interpolation
Solve, 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 Join[{1, 2}, Join[{3, 4}, {5}]].

3.10 Find a simpler form for Join[Range[10], Join[Range[10], Range[5]]].

3.11 Find a simpler form for Reverse[Join[Range[20], Reverse[Range[20]]]] (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 Join[Join[Range[10], Reverse[Range[10]]], 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]
```

$$\left\{2, \frac{5}{2}, 3, \frac{7}{2}, 4, \frac{9}{2}, 5, \frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, 8, \frac{17}{2}, 9\right\}$$

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.}];
```

$$\left\{2, \frac{5}{2}, 3, \frac{7}{2}, 4, \frac{9}{2}, 5, \frac{11}{2}, 6, \frac{13}{2}, 7, \frac{15}{2}, 8, \frac{17}{2}, 9\right\}$$

Digression (Solve, NSolve)

▪ 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] }
```

Digression (Array, Union, Pure Function, SetDelayed)

▪ 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, b, c];
{a, b, c} == {b, c, a}
{a, b, c} === {b, c, a}
```

```
{a, b, c} == {b, c, a}
```

```
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[x]
```

Reverse: Nonatomic expression expected at position 1 in Reverse[x].

```
Reverse[x]
```

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

```
True
```

```
True
```

```
True
```

```
dotProduct = a . b . c ;
Reverse[dotProduct]
```

```
c.b.a
```

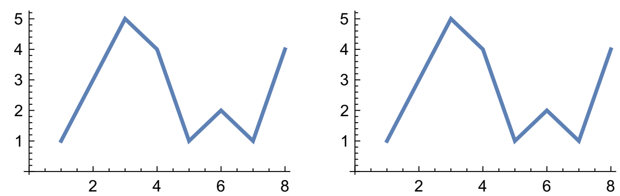
Displaying Lists

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.

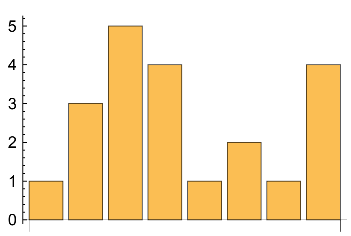
```
llp = ListLinePlot[{1, 3, 5, 4, 1, 2, 1, 4}, ImageSize → Small];
lp = ListPlot[{1, 3, 5, 4, 1, 2, 1, 4}, Joined → True, ImageSize → Small] ;
GraphicsRow[{llp, lp}]
(* ListLinePlot and ListPlot differ  in  PointSize    *)
(* ListLinePlot uses  Rational[7,360] ≈ 0.019444444444444445` *)
(* ListPlot uses Rational[77, 6000] ≈ 0.012833333333333334`*)
```



Making a **BarChart** can be useful too

Values give bar heights:

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

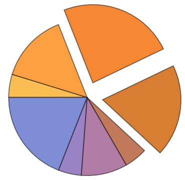


If the list is not too long, a **PieChart** can be useful

Values give the size of each slice (wedge).
Click on a slice to “explode” it .

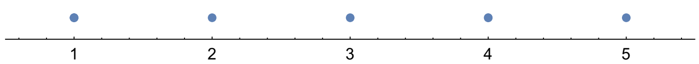
Slices have a relative sizes determined by the relative sizes of numbers in the list.
The slice for the first number starts at the 9 o’clock position; subsequent slices read clockwise.

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



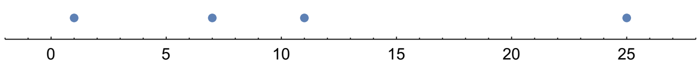
If you just want to know which numbers appear, you can **Plot** them on a **Number Line**

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

```
1
3
5
4
1
2
1
4
```

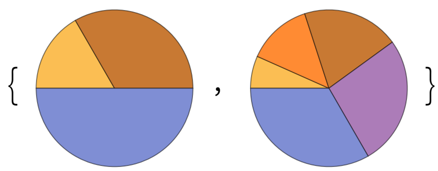
```
Column[{100, 350, 502, 400}]
```

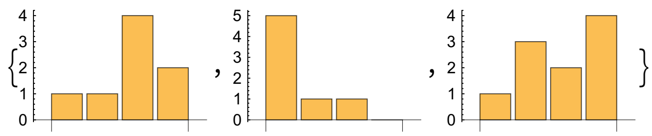
```
100
350
502
400
```

Combine plots

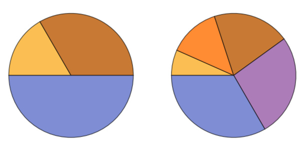
- Lists can contain anything, including Graphics.
  So, you can combine plots by putting them in lists.
- A list of plots can appear as the (input or) output of a computation, since *Mathematica* is symbolic.

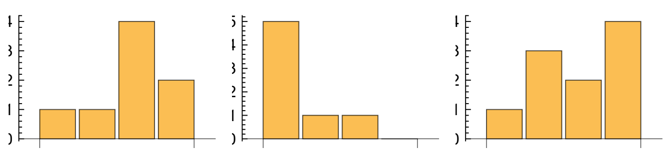
```
{PieChart[Range[3], ImageSize → Tiny],
 PieChart[Range[5], ImageSize → Tiny]}
{BarChart[{1, 1, 4, 2}, ImageSize → Tiny],
 BarChart[{5, 1, 1, 0}, ImageSize → Tiny],
 BarChart[{1, 3, 2, 4}, ImageSize → Tiny]}
```



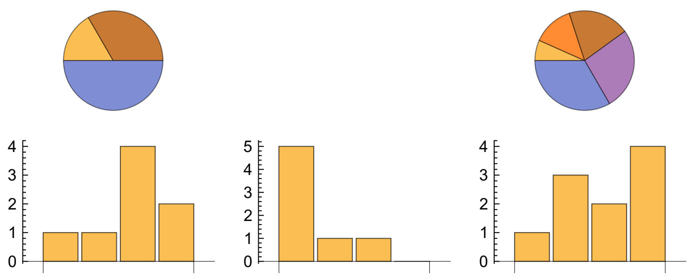



```
(* To  group graphics o images, it is better to use GraphicsRow or GraphicsGrid *)
GraphicsRow[{PieChart[Range[3]], PieChart[Range[5]]}, ImageSize → Small]
GraphicsRow[
 {BarChart[{1, 1, 4, 2}], BarChart[{5, 1, 1, 0}], BarChart[{1, 3, 2, 4}]}, ImageSize → Medium]
```





```
(* 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]}
 }]
```



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} .

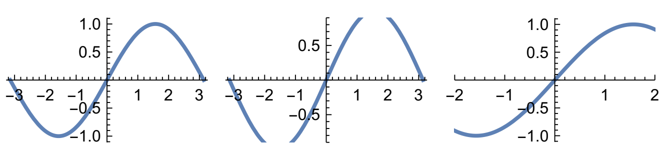
+4.6 Make a number line of fractions 1/2, 1/3, ... through 1/9 .

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]
```



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}
```

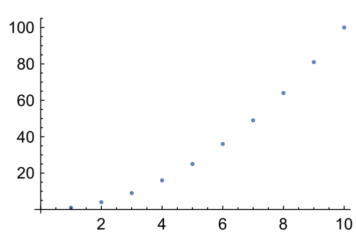
```
9
```

Range and **ListPlot**

Compute the first 10 squares, then ListPlot them

```
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 Integer, 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[%]
```

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

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

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

```
(* I cannot ask for this  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}
```

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

Digressione (vettori, matrici, tensori; Array, Funzione Pura)

Total gives the total from adding up a list

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

Find the total of the integers from 1 to 10 :

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

```
55
```

Count the number of times something appears in a List

```
Count[{a, b, a, a, c, b, a}, a]
```

```
4
```

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

```
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};
slist = Sort[blist] ;
First[ slist ] == 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};
(* Drop = scartare, buttare *)
{Rest[blist], Rest[blist] == Drop[blist, 1]}
{Most[blist], Most[blist] == Drop[blist, -1]}
```

```
{{7, 1, 2, 4, 5}, True}

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

```
(* La Part 0 di una lista e' la Head List *)blist = {6, 7, 1, 2, 4, 5};
blist[[0]]
TreeForm[blist, ImageSize → Small];
```

```
List
```

IntegerDigits makes a List of the digits in an Integer

The default is (base 10) decimal digits.
You can specify any base (e.g., binary or hexadecimal) :

```
IntegerDigits[203]
IntegerDigits[203, 2]
IntegerDigits[203, 16]
```

```
{2, 0, 3}

{1, 1, 0, 0, 1, 0, 1, 1}

{12, 11}
```

```
(* NOTE:  0203 == 203 *)
IntegerDigits[203] == IntegerDigits[0203]
```

```
True
```

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

```
IntegerDigits[o203]
```

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

```
{331, True}
```

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

```
{459, 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**

```
(* Take the first 3 elements from mylist *)
mylist = {11, 23, 41, 0, 62, 32, 12};
Take[mylist, 3]
```

```
{11, 23, 41}
```

```
(* Drop drops elements from the beginning of a list *)
Drop[mylist, 3]
```

```
{0, 62, 32, 12}
```

Vocabulary

{2,3,4}+{5,6,2}	arithmetics on lists
Sort[{5,7,1}]	sort a list into order
Length[{3,3}]	length of a list (number of elements)
Dimensions	
Total[{1,1,2}]	total of all elements in a list
Count[{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

Part[{3,1,4},2] part of a list, also written as {3, 1, 4}[[2]]
Take[{6,4,3,1},2] take elements from a list
Drop[{6,4,3,1},2] drop elements from a list
IntegerDigits[1234] list of digits in an integer
FromDigits[{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 & **Plus** to make a list of numbers from 10 to 20, inclusive.

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 Use Take & IntegerDigits to find the first 10 digits in 2^100.

5.10 Find the largest digit that appears in 2^20.

5.11 Use Count & IntegerDigits to find out 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 (and in 5^9).

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]**.

+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.

+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.

+5.7 Make a list of pie charts for the sequence of digits in 2^20, 2^40, 2^60.

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}
```