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 procedimeno di match-di-pattern, in *Mathematica*, fa si' 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 varieta' 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  *)
```

$$3\,a + 20.25\,b^2$$

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
```

$9\,a + 4.5\,b^4$

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
```

```
Plus^2[Times^2[3, a^2], Times^2[4.5, Power^2[b^2, 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}
```

```
Plus^2[Times^2[3, 9], Times^2[4.5, Power^2[4, 2]]]
```

Power^2[4, 2] non e' valutabile ed e' diverso da Power[4, 2]^2 (e lo stesso vale per Times):

```
Map[ FullForm, {Power^2[4, 2], Power[4, 2]^2 }]
```

```
{Power[Power, 2][4, 2], 256}
```

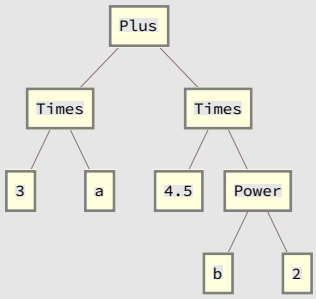
```
Power^2[4, 2] === Power[4, 2]^2
(* NOTA: Se si vuole eseguire la comparazione con Equal, invece che con SameQ,
e si vuole ottenere sempre un Booleano, si puo' usare  TrueQ *)
 TrueQ[Power^2[4, 2] == Power[4, 2]^2]
```

```
False
```

```
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 : sostituisco Plus  con  Times  *)
(* expr = 3 a + 4.5 b^2  *)
expr /. Plus → Times
ReplaceAll[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 :  sostituisco  a  con  x^2 ,
b con  x^2 per cui b^2 diventa x^4*)
(* 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 qualche esempio in cui useremo una variabile-pattern; definiamo l' espressione test :

```
Clear[expr, f, g, a, b];
test = (f[a] + g[b]) / f[a, b];
```

⌘ Nell'esempio 1 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 ⟶ 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 = (f[a] + g[b]) / f[a, b];
test /. expr_f → expr^2
(* Possiamo scrivere anche come segue: *)
test /. t_f → t^2 ;
```

$$\frac{f[a]^2 + g[b]}{f[a, b]^2}$$

⌘ Nell'esempio 2 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_] ⟶ x^2 .
Ne segue che f[a] viene sostituito con a^2 .

```
test = (f[a] + g[b]) / f[a, b];
test /. f[x_] → x^2
```

$$\frac{a^2 + g[b]}{f[a, b]}$$

⌘ Nell'esempio 3 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_] ⟶ (x+y)^2 .
Ne segue che f[a,b] e' sostituito da (a+b)^2 .

```
test = (f[a] + g[b]) / f[a, b];
test /. f[x_, y_] → (x + y)^2
```

$$\frac{f[a] + g[b]}{(a + b)^2}$$

```
(* Esempio 3bis *)
test = (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 sulla **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 4 che segue:

```
Clear[test, x, y];
test = x / Exp[y];
test // OutputForm
(*  OutputForm stampa una rappresentazione bidimensionale di expr,
come fa anche StandardForm,
ma usando solo i caratteri della tastiera*)test // StandardForm;
```

$$\frac{x}{E^y}$$

Usiamo la regola Exp[t_] → t
Se l'intento fosse stato quello di ottenere $\frac{x}{y}$, resteremmo sorpresi dal risultato della sostituzione:

```
test = x / Exp[y];
test /. Exp[t_] → t
```

```
-x y
```

Capiamo il motivo del risultato della sostituzione, esaminando la forma interna (FullForm) di **test** e del pattern :

```
Map[FullForm, {test, Exp[t_]}]
```

```
{Times[Power[E, Times[-1, y]], x], Power[E, Pattern[t, Blank[]]]}
```

```
(* La Forma Interna di  { test ,Exp[t_ ] }  e'  quella qui  sotto  *)
{ Times[Power[E, Times[-1, y]], x] , Power[E, Pattern[t, Blank[]]] }
```

$$\{e^{-y} x, e^{t\_}\}$$

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^[-y] → -y** ovvero **Exp[-y] → -y**

```
{ Power[E , Times[-1, y]] , Times[-1, y] }
```

$\{e^{-y}, -y\}$

Dunque otteniamo:

```
test = x / 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 sotto-espressioni.
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.

```
Clear[expr];
expr = f[a] + g[b];
expr // 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):

```
(* expr=f[a]+g[b] *)
expr /. 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] :

```
(* expr=f[a]+g[b] *)
expr /. x_[_, _] → x
```

```
Plus
```

L'esempio seguente illustra che e' possibile fare praticamente qualsiasi cosa, mediante de-strutturazione.

```
(* expr=f[a]+g[b] *)
expr /. 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
```

$$\begin{pmatrix} x1 & y1 \\ x2 & y2 \\ x3 & y3 \\ x4 & y4 \end{pmatrix}$$

Separo i valori x ed y

```
tdata = Transpose[data]
tdata // MatrixForm
```

```
{{x1, x2, x3, x4}, {y1, y2, y3, y4}}
```

$$\begin{pmatrix} x1 & x2 & x3 & x4 \\ y1 & y2 & y3 & y4 \end{pmatrix}$$

Uso **MapAt** per trasformare i valori **y** in scala logaritmica.
Prima ricordo come funziona MapAt :

```
? MapAt
```

Symbol

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_1, j_1, ...}, {i_2, j_2, ...}, ...}] 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]
(* Equivalente : MapAt[f,{a,b,c,d},{2}] *)
```

```
{a, f[b], c, d}
```

```
(* Applico  f  alle posizioni 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  alla 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 la matrice trasposta **tdata**.
Uso **MapAt** per trasformare i valori **y** (contenuti nella Parte 2 di **tdata**) in scala logaritmica.
Questa operazione sfrutta il fatto che Log e' Listable (cfr. 3.3).

```
tdata
(* Applico  Log  alla posizione 2  della matrice  tdata,
ossia alla sua seconda riga *)
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]}}
```

$$\begin{pmatrix} x1 & x2 & x3 & x4 \\ \text{Log}[y1] & \text{Log}[y2] & \text{Log}[y3] & \text{Log}[y4] \end{pmatrix}$$

Ricombino i valori **x** ed i valori **Log[y]**

```
tmtdata = Transpose[mtdata]
tmtdata // MatrixForm
```

```
{{x1, Log[y1]}, {x2, Log[y2]}, {x3, Log[y3]}, {x4, Log[y4]}}
```

$$\begin{pmatrix} x1 & \text{Log}[y1] \\ x2 & \text{Log}[y2] \\ x3 & \text{Log}[y3] \\ x4 & \text{Log}[y4] \end{pmatrix}$$

```
(* Ricapitolando: *)
data = {{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}};
tdata = Transpose[data];
mtdata = MapAt[Log, tdata, 2];
tmtdata = Transpose[mtdata];
tmtdata // MatrixForm
```

$$\begin{pmatrix} x1 & \text{Log}[y1] \\ x2 & \text{Log}[y2] \\ x3 & \text{Log}[y3] \\ x4 & \text{Log}[y4] \end{pmatrix}$$

Il procedimento qui sopra puo' essere eseguito piu' elegantemente coi pattern (pattern matching)

```
data = {{x1, y1}, {x2, y2}, {x3, y3}, {x4, y4}};
datafast = data /. {x_, y_} → {x, Log[y]}
datafast // MatrixForm
```

```
{{x1, Log[y1]}, {x2, Log[y2]}, {x3, Log[y3]}, {x4, Log[y4]}}
```

$$\begin{pmatrix} x1 & \text{Log}[y1] \\ x2 & \text{Log}[y2] \\ x3 & \text{Log}[y3] \\ x4 & \text{Log}[y4] \end{pmatrix}$$

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]}}
```

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 combacera'.

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] *)
(* Provare anche con :exprLista2={a,a+b,a+a, ab+ba, a+b+c } *)
```

```
{a, a + b, 2 a}
```

```
x_ + y_
```

```
{a + b}
```

Uso FullForm per capire l'output di Cases;
qui il pattern e' Plus[x_ , y_] ,
quindi Cases estrae un elemento da exprLista solo se tale elemento ha Head **Plus** e due argomenti **x_** ed **y_**

```
(* exprLista={a,a+b,a+a};   pattern=x_+y_ ;  *)
Map[FullForm, {exprLista, pattern}] // TableForm
```

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

```
True
```

Un altro esempio.

```
listaH = {a, a + b, HoldForm[a + a]}
listaH // FullForm
(* Il secondo elemento di listaH  combacia col pattern  x_+y_  *)
Cases[listaH, x_ + y_]
(* Il terzo elemento di listaH  combacia col pattern  _[x_+y_]  *)
Cases[listaH, _[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).

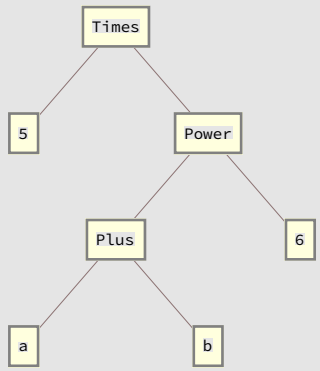
```
lista = {a, a + b, a + a};
(* Cases  estrae  a+b  da  lista *)
Cases[lista, x_ + y_]
(* Cases  estrae  a+b  da  lista  e  lo  sostituisce  con  b *)
Cases[lista, x_ + y_ → y]
```

```
{a + b}
```

```
{b}
```

Note.
La Head del primo argomento di Cases non deve essere necessariamente List .
Di default, Cases lavora SOLO al livello 1.

```
exprNonLista = 5 (a + b)^6;
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 estrae tutti gli interi di exprNonLista, a qualsiasi livello *)
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
(* Specificando Heads→True, Cases estrae anche le Head *)
Cases[exprNonLista, _Symbol, Infinity, Heads → True]
```

```
{a, b}
```

```
Times[5, Power[Plus[a, b], 6]]
```

```
{Times, Power, Plus, a, b}
```

Colors and Styles

In *Mathematica* , you can handle various things (not just numbers), e.g., colors.

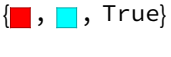
You can refer to common colors by their names.

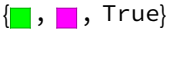
```
{Red, Green, Blue, Purple, Orange, Black}
(* swatch *)
```

You can do operations on colors.

▪ **ColorNegate** gives the complementary color of a specified color,
defined by computing 1–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 }
```





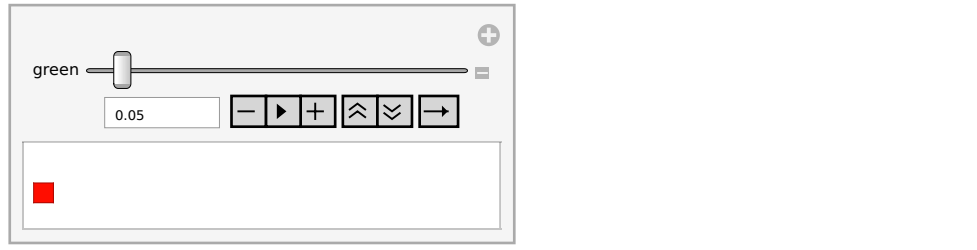
▪ **Blend** blends a list of colors together.

```
Blend[{Yellow, Pink, Green}]
(* Mescola = 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[h] is equivalent to Hue[h,1,1].

```
{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]
```

```
1000
```

```
(* 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"]}
```

$\{x, \mathbf{x}, x, \mathit{x}\}$

```
(* 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 5 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

$FontFamilies

Import, Export, **$ImportFormat**, **$ExportFormat**

Manipulate

GrayLevel

MapThread

Cases, Except, Alternatives

Nest, NestList
Thread

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 3 even numbers (starting with 4) **squared**, in which each value is styled at its size (**funzione definita su 1, poi 2 variabili**) .

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, α, β] or LABColor[{L, α, β}],
where L is lightness (brightness)
and α, β are color components (respectively, Green to Magenta, Blue to Yellow) .
You can specify the Optional argument ω opacity: LABColor[L, α, β, ω]; default is ω = 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 ]}
```

{■, ■, ■, ■}

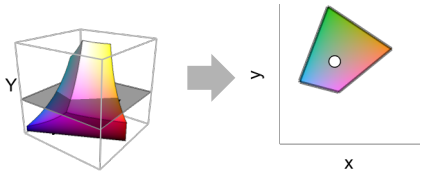
You can specify named HTML colors (e.g. RGBColor["aqua"]) as well as hex colors (e.g. RGBColor["#00ff00"])

```
{ RGBColor["aqua"],    RGBColor["aqua"] == Cyan,
 RGBColor["#00ff00"], RGBColor["#00ff00"] == Green}
```

{■, True, ■, True}

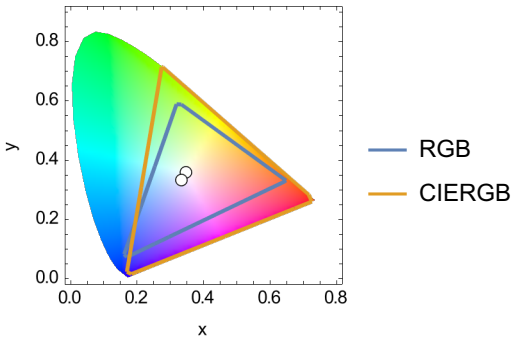
ChromaticityPlot and **ChromaticityPlot3D** plot lists of colors in color space.

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).

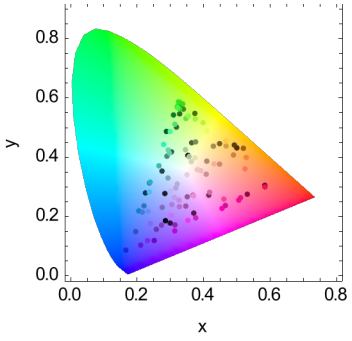


→ It can be used to compare to color space models.

```
(* 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**.

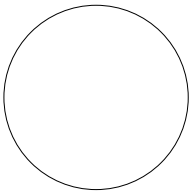
Basic Graphics Objects

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]
```

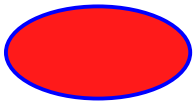
Use the graphics transparency directive **Opacity :**

```
obj1 = {Opacity[0.3],
    Disk[{0, 0}, {2, 1}],
    Disk[{2, 2}, {1, 1}]};
g1 = Graphics[obj1];

obj2 = Style[
    { Disk[{0, 0}, {2, 1}],  Disk[{2, 2}, {1, 1}] },
    Opacity[0.3]];
g2 = Graphics[obj2];

GraphicsRow[{g1, g2}, Spacings → 50, ImageSize → Small]
```

```
(* FullForm[g1]
 FullForm[g2] *)
```

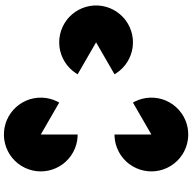
Use the graphics directive **EdgeForm :**

```
objEF = { EdgeForm[{Thick, Blue}],
    Opacity[0.9],
    Red,
    Disk[{0, 0}, {2, 1}] };
Graphics[ objEF, ImageSize → Tiny]
(* Red is equivalent to RGBColor[1,0,0]  *  *)
(*  RGBColor  is  a  graphics directive *)
(* ImageSize  is  an  Option  *)
```



Create illusory contours :

```
illusion = { 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}]};
Graphics[illusion, ImageSize → Tiny]
```



Make an oval pie-chart (an example of use of Module) :

```
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} *)

(* Esempi di Documentazione di un codice  *)
(* pie e' una funzione che fa  questo ...  *)
(* data e' una lista di input e contiene questo ... *)
(* descrizione dell' output *)
pie[data_] := Module[
  {t = 0, xc = 0, yc = 0, rx = 2, ry = 1, len, sum, dataNorm, paramOpacity = 0.8, sectors},
  (* descrizione delle variabili di lavoro *)
  (* t:  angle spanning sectors in the pie-chart *)
  (* xc,yc: center of the pie-chart *)
  (* rx,ry: xradius and yradius of the oval pie-chart *)
  (* len: length of data *)
  (* sum = sum of all data *)
  (* dataNorm : data normalized in [0,1] *)
  (* paramOpacity: parameter for Opacity used later, in Graphics *)
  (* sectors : table of sectors forming the oval pie *)

  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 += m  pre-incremental updating  x=x+m  *)
  ];

  (* Define and render sectors forming the oval pie-chart *)
  sectors = Table[
    {
     (*  k/len  is  in [0,1] *)
     Hue[k / len],
     EdgeForm[Opacity[paramOpacity]],
     (* A sector is defined as Disk[{xc,yc}, {r_x,r_y},  {t_{k-1},t_k}] *)
```

```
      Disk[ {xc, yc}, {rx, ry}, {t, t += 2 Pi  dataNorm[[k]] } ]
     },
     {k, len}] ;  (* end Table *)

  Graphics[sectors, ImageSize → Tiny]
 ](* end Module *)
```

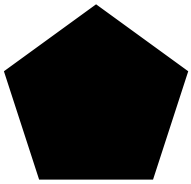
```
pie[data]
```

```
{{0, 2.39153}, {2.39153, 4.70154}, {4.70154, 5.40771}, {5.40771, 5.95719}, {5.95719, 6.28319}}
```

NOTE 1. Operators of (pre) increment / decrement

RegularPolygon[n] gives a regular polygon with n > 2 sides .

```
(* pentagon *)
Graphics[RegularPolygon[5], ImageSize → Tiny]
```



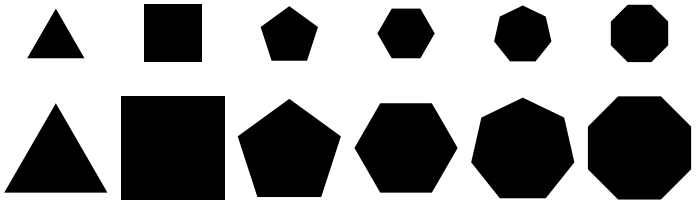
You can find a **list** of REGULAR POLYGONS NAMES , for instance, at
https : // www.mathsisfun.com/geometry/polygons.html

```
(*  Hyperlink["https://www.mathsisfun.com/geometry/polygons.html"] *)
```

```
(* gr is a list made of:
  triangle or trigon, quadrilateral or tetragon,
pentagon, hexagon, heptagon, octagon *)
gr = Table[
    Graphics[RegularPolygon[n], ImageSize → Tiny],
    {n, 3, 8}];
GraphicsRow[gr, ImageSize → Medium];

(* Another way, using Map  *)
rp = Table[ RegularPolygon[n], {n, 3, 8}];
mgr = Map[Graphics, rp];
gr2 = GraphicsRow[ mgr, ImageSize → Medium ]
```

```
(* Different sizes *)
gr3 = Table[
    Graphics[  RegularPolygon[3] , ImageSize → s],
    {s, {Tiny, Small}}];
GraphicsRow[ gr3, ImageSize → Tiny]
```

```
(* One Table with two iterators *)
grs = Table[
    Graphics[RegularPolygon[n], ImageSize → s],
    {s, {Tiny, Small}},
    {n, 3, 8}
  ];
(* TableForm[grs]  *)
GraphicsGrid[ grs , ImageSize → Medium ]
```



```
(* Another Table with the two iterators swapped *)
grs = Table[
    Graphics[RegularPolygon[n], ImageSize → s],
    {n, 3, 8},
    {s, {Tiny, Small}}
  ];
GraphicsGrid[ grs ];
```

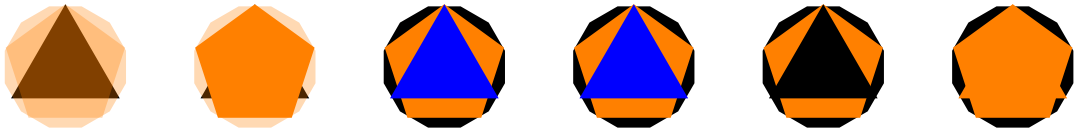
Style works inside **Graphics**, so you can use it to give colors.

```
(*
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];

p0 = {Orange, penta};
(* List[ RGBColor[1,0.5`,0], RegularPolygon[5] ] *)
(* g0=Graphics[p0]; *)

p1 = {tria, (* default color is Black *)
    Orange, Opacity[0.3], penta, dodeca};
g1 = Graphics[p1];

p2 = {tria,
    Orange,
    (* qui, grazie a List, Opacity ha effetto solo su dodeca *)
    {Opacity[0.3], dodeca},
    penta};
g2 = Graphics[p2];

p3 = {dodeca,
    p0, (* {Orange,penta} *)
    Blue, tria};
g3 = Graphics[p3];
```

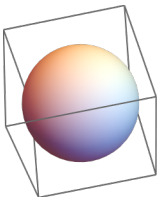
```
(*  Flatten, di default, appiattisce tutti i livelli,
restituendo una lista mono-dimensionale *)
p4 = Flatten[p3];
g4 = Graphics[p4];
(* Flatten mostra che e' superfluo, in p3, usare List attorno a { Orange, penta } *)
FullForm[p3];
FullForm[p4];

(* Qui, invece, Flatten serve,  per modificare g5 in g6 *)
p5 = {dodeca,
    p0, (* {Orange,penta} *)
    tria};
g5 = Graphics[p5];

p6 = Flatten[p5];
g6 = Graphics[p6];
FullForm[p5];
FullForm[p6];

GraphicsRow[{g1, g2, g3, g4 , g5,  g6}, Spacings → 50]
```



Sphere, **Cylinder** , **Cone** are 3D constructs, that can be rendered with **Graphics3D**.

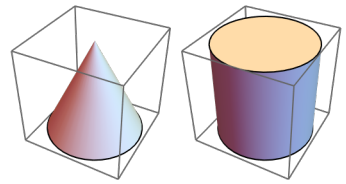
You can rotate 3D graphics interactively to see different angles.

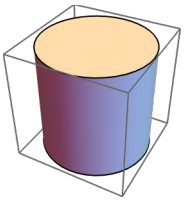
```
Graphics3D[
  Sphere[],
 ImageSize → Tiny]
```

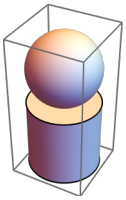


- A **list** of Graphics3D directives (Cone and Cylinder).
- One Graphics3D, whose argument is a list of graphics objects (Sphere and Cylinder) .

```
g3d = { Graphics3D[ Cone[]],
    Graphics3D[Cylinder[]] };
(* ccl={ Cone[],  Cylinder[]  };
   g3d=Map[  Graphics3D, ccl  ];  *)
GraphicsRow[g3d, ImageSize → Small]

(* Qui, la sfera viene resa interna al cilindro, quindi non si vede *)
Graphics3D[
 { Sphere[], Cylinder[] },
 ImageSize → Tiny]

(* Sfera esterna al cilindro *)
Graphics3D[
 { Sphere[{0, 0, 2}], Cylinder[] },
 ImageSize → Tiny]
```

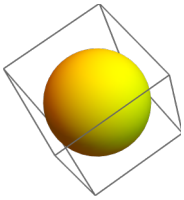






A yellow sphere; note that it is rendered like an actual 3D object, with lighting;
if it were pure yellow, we would not see any 3D depth, and it would look like a 2D disk.

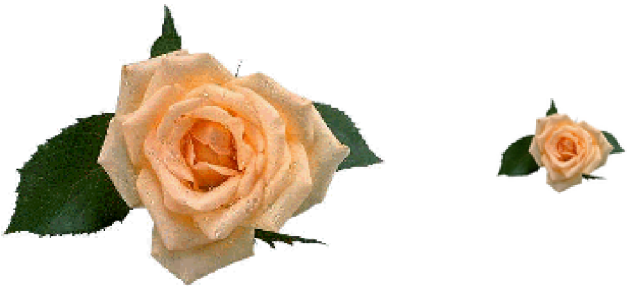
```
Graphics3D[
 Style[Sphere[], Yellow],
 ImageSize → Tiny]
```



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
```

NOTE 2. Setting the working Directory

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	
Hyperlink	
Flatten	
Directory[]	
SetDirectory[]	
NotebookDirectory[]	
AppendTo[]; Append[]	
If, Which, Switch, Piecewise, Cases	

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 *

6.1.4 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)

```
Map[Attributes, {Plus, Times (* , Power,Cases *)}] // TableForm
```

```
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).

OneIdentity 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. Plus[a, b] == Plus[b , a]

Protected significa che non e' possibile definire nuove regole per Plus senza prima usare Unprotect (cfr. 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 *)
```

```
? Flat
(* proprieta'  associativa  *)
```

```
? NumericFunction
(* Eg:  NumericQ[Log[2]]  returns  True  *)
```

```
? OneIdentity
(* per Plus significa che Plus[x]== x *)
```

```
? Orderless
(* proprieta'  commutativa  *)
```

```
? Protected
```

```
a + b + c + d == (a + b) + (c + d)
```

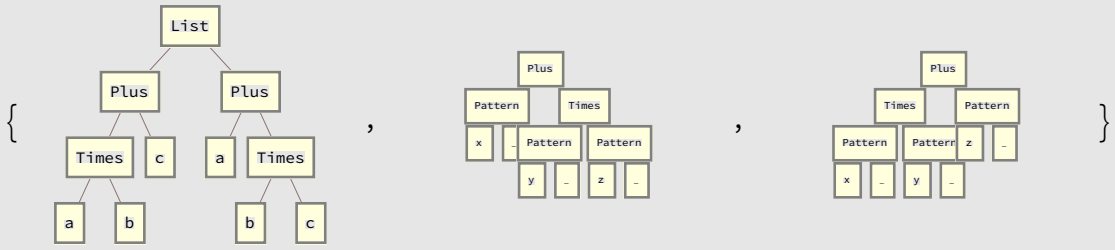
```
True
```

```
Map[Attributes, {Plus, Times}] // TableForm
```

```
Flat    Listable    NumericFunction    OneIdentity    Orderless    Protected
Flat    Listable    NumericFunction    OneIdentity    Orderless    Protected
```

⌘ L'attributo (di commutativita') Orderless fa si' che, nell'esempio qui sotto, il pattern x_+ y_* z_ combaci con entrambe le sotto-espressioni di expr.
Idem per il patternB x_* y_+ z_

```
Clear[expr];
(* expr={a+b*c, a*b+c};   *)
expr = {c + b * a, c * b + a};
(* Priorita'  tra  Times  e  Plus :
  Times viene eseguita prima di Plus ,
quindi Times si trovera' ad un livello non-zero,  verso  le  "foglie"  ,
mentre  Plus  e'  a  livello  zero *)
pattern = x_ + y_ * z_;
patternB = x_ * y_ + z_;
Map[TreeForm, {expr, pattern , patternB }]
(* sia pattern che patternB intercettano entrambe le sotto-espressioni di expr *)
Cases[expr, pattern]
Cases[expr, patternB]
(* Il riordinamento canonico, alfanumerico,e' visibile nelle stampe dell'I/O *)
```




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

{a b + c, a + b c}

```
(* Nota: valgono le seguenti uguaglianze *)
{pattern = x_ + y_ * z_;
 pattern2 = x_ + z_ * y_;
 pattern3 = y_ * z_ + x_;
 pattern4 = z_ * y_ + x_;
 pattern == pattern2 == pattern3 == pattern4,
 patternB = x_ * y_ + z_;
 patternB2 = y_ * x_ + z_;
 patternB3 = z_ + x_ * y_;
 patternB4 = z_ + y_ * x_ ;
 patternB == patternB2 == patternB3 == patternB4}
```

```
{True, True}
```

⌘ La possibilita' di ottenere comportamenti quali quelli visti qui sopra permette di realizzare trasfor-

mazioni anche sofisticate, con poco sforzo.
Ad esempio, la regole 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)** .

```
Clear[expr];
expr = a (b + c) (d + e + f)
```

```
a (b + c) (d + e + f)
```

Passo**1**. 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).

```
(* expr = a (b+c) (d+e+f); *)
(* expandrule = x_ * (y_+z_) → x * y + x * z ; *)
passo1 = expr /. expandrule
(* y_ + z_  combacia con  b+c
  mentre x_ combacia con  a(d+e+f)  *)
```

```
a b (d + e + f) + a c (d + e + f)
```

Passo**2**. Il pattern **y_ + z_** combacia con **d + (e+f)**

```
(* passo1 = a b (d+e+f) + a c (d+e+f); *)
(* expandrule = x_ * (y_+z_) → x * y + x * z ; *)
passo2 = passo1 /. expandrule
(* 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) *)
```

```
a b d + a c d + a b (e + f) + a c (e + f)
```

Passo**3**. Il pattern **y_ + z_** combacia con **e+f**

```
(* passo2 = a b d + a c d + a b (e+f)+a c (e+f); *)
(* expandrule = x_ * (y_+z_) → x * y + x * z ; *)
passo3 = passo2 /. expandrule
(* 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) *)
```

```
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 *)
expr = a (b + c) (d + e + f)
(* y_ + z_  combacia con b+c ;  x_ combacia con a(d+e+f)  *)
passo1 = expr /. 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 FixedPoint.

FixedPoint[ ] applica una regola ad una espressione ripetutamente, fino a che l'espressione smette di cambiare (si arriva ad un Punto Fisso).

```
? FixedPoint
```

Symbol

FixedPoint[*f*, *expr*] starts with *expr*,
then applies *f* repeatedly until the result no longer changes.

```
expandrule = x_ (y_ + z_) → x y + x z;
expr1 = a (d + e + f);
expr  = a (b + c) (d + e + f);

(* definisco una funzione pura
    Function[ # /.expandrule ]   ovvero    #/.expandrule &
 ed
 uso FixedPoint *)

result1 = FixedPoint[ # /. expandrule &, expr1];
{expr1, result1}

result = FixedPoint[ # /. expandrule &, expr];
{expr, result}
```

$\{a\,(d + e + f),\ a\,d + a\,e + a\,f\}$

$\{a\,(b + c)\,(d + e + f),\ a\,b\,d + a\,c\,d + a\,b\,e + a\,c\,e + a\,b\,f + a\,c\,f\}$

⌘ 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;
expr1 = a (d + e + f);
expr  = a (b + c) (d + e + f);

{expr1,
 expr1 //. expandrule}
(* ReplaceRepeated[espr1,expandrule] *)

{expr,
 expr //. 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 gia' usato alcune **funzioni che accettano pattern come argomenti** (ad esempio, MatchQ e Cases, e anche Select).
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.

```
? Position
```

Symbol

Position[*expr*, *pattern*] gives a list of the
positions at which objects matching *pattern* appear in *expr*.
Position[*expr*, *pattern*, *levelspec*] finds
only objects that appear on levels specified by *levelspec*.
Position[*expr*, *pattern*, *levelspec*, *n*] gives the positions of the first *n* objects found.
Position[*pattern*] represents an operator
form of Position 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]
```

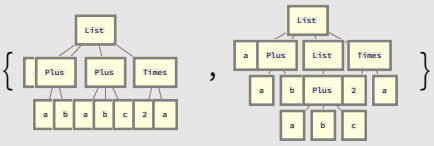
```
{a + b, a + b + c}
```

```
2
```

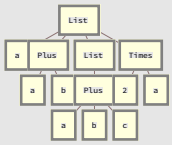
```
{{2}, {3}}
```

Come molte altre funzioni che accettano pattern, alle funzioni Count e Position puo' essere inoltre specificato un livello, per circostanziare la loro ricerca.

```
(* La lista expr2  e'  piu' annidata  di  expr *)
(* expr={a,a+b,a+b+c,a+a}; *)
expr2 = {a, a + b, {a + b + c},  a + a};
{TreeForm[expr, ImageSize → Tiny], TreeForm[expr2, ImageSize → Tiny]}
```



```
patt = x_ + y_;
expr2 = {a, a + b, {a + b + c}, a + a};
TreeForm[expr2, ImageSize → Tiny]
(* Default: Individua i Match al livello 1 *)
Cases[expr2, patt]
Cases[expr2, patt] == Cases[expr2, patt, 1];
(* Individua i Match fino al livello 2 *)
Cases[expr2, patt, 2]
(* Individua i Match solo al livello 2 *)
Cases[expr2, patt, {2}]
(* Individua i Match su tutti i livelli *)
Cases[expr2, patt, Infinity]
```



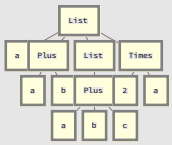

```
{a + b}
```

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

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

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

```
patt = x_ + y_;
expr2 = {a, a + b, {a + b + c}, a + a};
TreeForm[expr2, ImageSize → Tiny]
(* Default:  Numero di Match al livello 1 *)
Count[expr2, patt]
Count[expr2, patt] == Count[expr2, patt, 1];
(* Numero di Match fino al livello 2 *)
Count[expr2, patt, 2]
(* Numero di Match solo al livello 2 *)
Count[expr2, patt, {2}]
(* Numero di Match su tutti i livelli *)
Count[expr2, patt, Infinity]
```



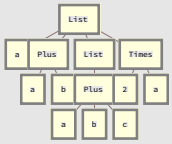
1

2

1

2

```
patt = x_ + y_;
expr2 = {a, a + b, {a + b + c}, a + a};
TreeForm[expr2, ImageSize → Tiny]
(* Posizione dei Match  a livello 1 :
 posizione di  Plus[a,b]  *)
Position[expr2, patt, 1]
(* Posizione dei Match fino al livello 2 :
 posizione  di  Plus[a,b]  e  di  Plus[a,b,c]  *)
Position[expr2, patt, 2]
(* Posizione dei Match solo al livello 2 :
 posizione  di Plus[a,b,c]  *)
Position[expr2, patt, {2}]
(* Default : Posizione dei Match su tutti i livelli  *)
Position[expr2, patt]
Position[expr2, patt] == Position[expr2, patt, Infinity];
```



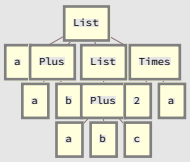
{{2}}

{{2}, {3, 1}}

{{3, 1}}

{{2}, {3, 1}}

```
TreeForm[expr2, ImageSize → Tiny]
(* NOTA. Ulteriore differenza nei default di Cases/Count versus Position :
   Position ha Heads → True come default:
   indica che, di default, le Head sono incluse nella ricerca *)
Position[expr2, List] == Position[expr2, List, {0, Infinity}, Heads → True];
Print["Position (Heads->True) default: ",
 Position[expr2, List],
 " ,   Position (Heads->False): ",
 Position[expr2, List, Heads → False]];

Cases[expr2, List] == Cases[expr2, List, Heads → False];
Print["Cases (Heads->False) default: ",
  Cases[expr2, List],
  " ,   Cases (Heads->True): ",
  Cases[expr2, List, Heads → True]];

Count[expr2, List] == Count[expr2, List, Heads → False];
Print["Count (Heads->False) default: ",
  Count[expr2, List],
  " ,    Count (Heads->True): ",
  Count[expr2, List, Heads → True]];
```




```
Position (Heads->True) default: {{0}, {3, 0}} ,   Position (Heads->False): {}
Cases (Heads->False) default: {} ,   Cases (Heads->True): {List}
Count (Heads->False) default: 0 ,    Count (Heads->True): 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
```

```
patt = x_ + y_;
expr = {a, a + b, a + b + c , a + a};
Cases[expr, patt]
DeleteCases[expr, patt]
```

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

```
{a, 2 a}
```

Secondo Esempio di DeleteCases

Terzo Esempio di DeleteCases

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, pero', 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**.

```
? 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

DownValues[*f*] gives a list of transformation rules
corresponding to all downvalues (values for *f*[...]) defined for the symbol *f*.
DownValues["*symbol*"] gives a list of transformation rules corresponding
to all downvalues defined for the symbol named "*symbol*" if it exists.

```
? HoldPattern
```

Symbol

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[ ].

```
Clear[f, g1, g2, x];
(* Come viene valutata f ?
      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];
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 specificita'/generalita') .

```
? 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 ≤ -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 le corrispondenze 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
```

Symbol

Global`f

Definitions

```
f[x_ /; x < 2] := g2[x]

f[x_ /; x > -2] := g1[x]
```

Full Name Global`f

```
(* Come viene valutata 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   *)
```

Ora **f** ha come definizioni multiple: se x < 2 allora g2; se x ≥ 2 allora g1 .
Ri-studiamo **f** sulle ascisse { -3, -2, -1, 0, 1, 2, 3 }.

```
(* Ricreo 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 intepretata NON una assegnazione (ad f) differita,
bensi' una regola differita con output x_^2 :↦ x^2 *)
f[x_] :↦ x^2
```

$$\{\text{HoldPattern}[f[x\_]] :\mapsto x^2\}$$

$$x\_^2 :\mapsto x^2$$

Osserviamo la differenza:

```
(* fa e' definito con una Assegnazione differita *)
fa[x_] := x^2; fa[x] // TraditionalForm
fa[x] /. x → 0
```

$$x^2$$

```
0
```

```
(* fr e' definito con una Regola differita *)
fr[x_] :↦ x^2; fr[x]
fr[x] /. x → 0
```

```
fr[x]
```

```
fr[0]
```

Sembra che al simbolo **fr** non venga associato nulla.
In effetti, e' cosi', e lo possiamo vedere bene con DownValues:

```
Clear[fa, fr];
fa[x_] := x^2; DownValues[fa] // TraditionalForm
fr[x_] :↦ x^2; DownValues[fr]
```

$$\{\text{HoldPattern}[\text{fa}(x\_)] :\mapsto x^2\}$$

```
{}
```

Nota. Il fatto che ad **fr** non viene associato nulla viene segnalato anche dai colori (nero per **fa**, blu per **fr**)

```
? fa
? fr
```

Symbol

Global`fa

Definitions

$fa[x\_] := x^2$

Full Name Global`fa

Symbol

Global`fr

Full Name Global`fr

Dato che (a differenza di quanto accade per **fa**) 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
fa[trr]
Map[fa, trr]
(* trr^2 == Map[fa,trr] *)
fr[trr]
Map[fr, trr]
```

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

```
{4, 25, 9, 9, 9, 25, 16, 25, 16, 16}
```

```
{4, 25, 9, 9, 9, 25, 16, 25, 16, 16}
```

```
fr[{2, 5, 3, -3, 3, -5, 4, 5, 4, -4}]
```

```
{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

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 piu'.

```
? ReplaceRepeated
expr = x^2 + y^6;
expr //. {x → 2 + a, a → 3} // TraditionalForm
```

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.

$$y^6 + 25$$

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 RepleaceRepeated.
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]
```

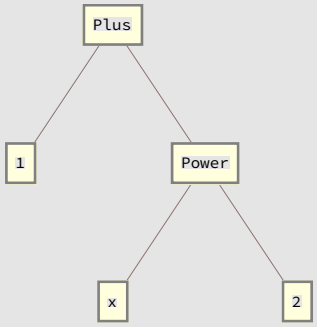
```
log[a] + log[b c d]
```

```
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 *)
TreeForm[1 + x^2, ImageSize → Small]
Replace[1 + x^2, x^2 → a + b] (* Il livello di default per Replace e' {0} *)
Replace[1 + x^2, x^2 → a + b, {1}]
? Replace
```



$1 + x^2$

$1 + a + b$

Symbol

Replace[*expr*, *rules*] applies a rule or list of
rules in an attempt to transform the entire expression *expr*.
Replace[*expr*, *rules*, *levelspec*] applies rules to parts of *expr* specified by *levelspec*.
Replace[*rules*] represents an operator form of Replace that can be applied to an expression.

```
(* 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

ReplacePart[*expr*, $i \to new$] yields an
expression in which the i^{th} part of *expr* is replaced by *new*.
ReplacePart[*expr*, {$i_1 \to new_1$, $i_2 \to new_2$, ...}] replaces parts at positions i_n by new_n.
ReplacePart[*expr*, {i, j, ...} → *new*] replaces the part at position {i, j, ...}.
ReplacePart[*expr*, {{i_1, j_1, ...} → new_1, ...}] replaces parts at positions {i_n, j_n, ...} by new_n.
ReplacePart[*expr*, {{i_1, j_1, ...}, ...} → *new*] replaces all parts at positions {i_n, j_n, ...} by *new*.
ReplacePart[$i \to new$] 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

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}}}

Interactive Manipulation

So far, we have been using *Mathematica* in a question-and-answer way: we type an input and obtain an output.

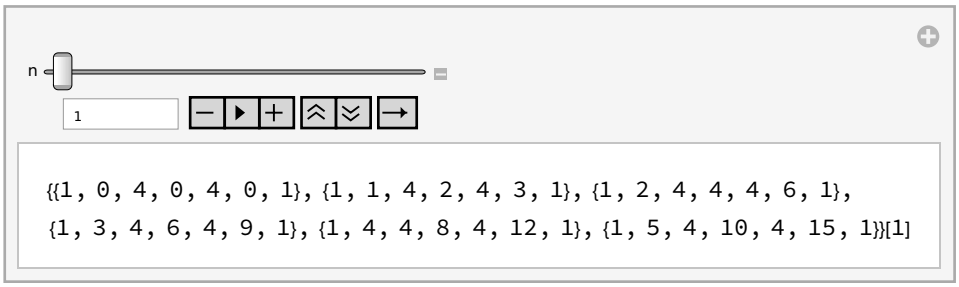
The built-in **Manipulate** serves to create an interface where we can continually manipulate one or more variables (parameters).

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}]
```

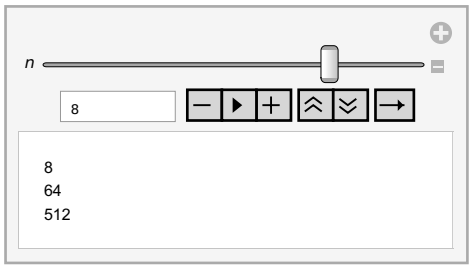
{{■}, {■, ■}, {■, ■, ■}, {■, ■, ■, ■}, {■, ■, ■, ■, ■}}

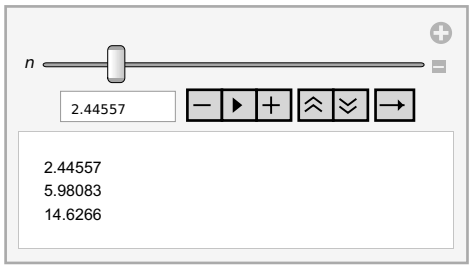


By default, Table assumes **start=step=1**.
Manipulate needs **start** to be specified.
If you omit the step size, Manipulate uses a non-integer step
(it assumes you want machine-precision values, in the specified range).

```
(* Importance of start and stepsize in Manipulate *)
cc[n_] := Column[{n, n^2, n^3}];
Table[cc[n], {n, 10}]
GraphicsRow[{Manipulate[cc[n], {n, 1, 10, 1}],
  Manipulate[cc[n], {n, 1, 10}]}]
```

```
 1   2   3    4    5    6    7    8    9    10
{1 , 4 , 9  , 16 , 25 , 36 , 49 , 64 , 81 , 100 }
 1   8   27   64   125  216  343  512  729  1000
```



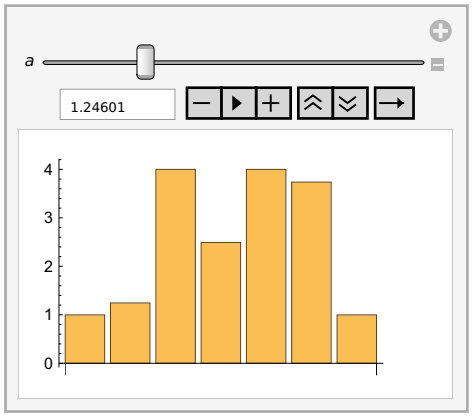


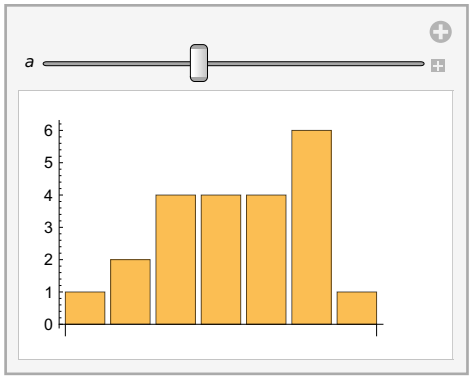

```
ff[n_] := Column[Map[FullForm, {n, n^2, n^3}]];
Manipulate[ff[n], {n, 1, 10}];
```

Manipulate and Charts

A bar chart that changes as you move the slider:

```
GraphicsRow[{
  Manipulate[
   BarChart[ {1, a, 4, 2*a, 4, 3*a, 1}, ImageSize → Small],
   {a, 0, 5}],

  (* Inizialize  to  "2"  the  "a" parameter in the slider *)
  Manipulate[
   BarChart[ {1, a, 4, 2*a, 4, 3*a, 1}, ImageSize → Small],
   {{a, 2}, 0, 5}]
 }]
```



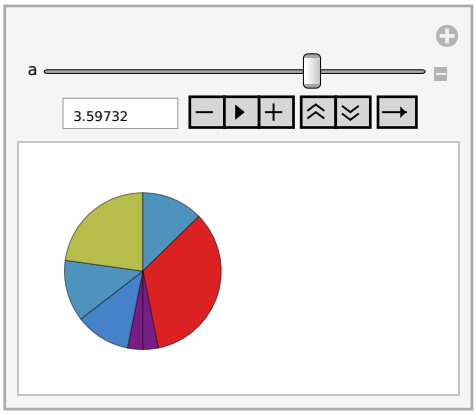


A pie chart that changes as you move the slider :

```
tt = Table[ {1, a, 4, 2*a, 4, 3*a, 1},
   {a, 0, 5}]
```

```
{{1, 0, 4, 0, 4, 0, 1}, {1, 1, 4, 2, 4, 3, 1}, {1, 2, 4, 4, 4, 6, 1},
 {1, 3, 4, 6, 4, 9, 1}, {1, 4, 4, 8, 4, 12, 1}, {1, 5, 4, 10, 4, 15, 1}}
```

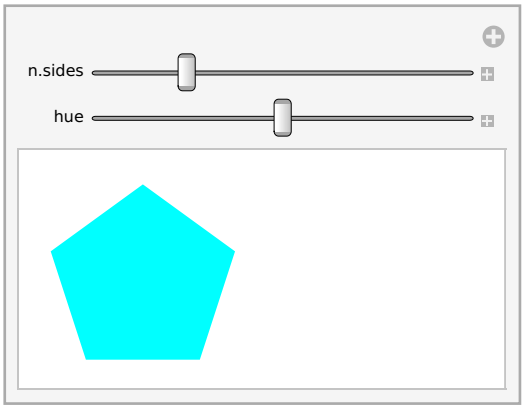
```
(* PieChart[{y0,y1,…,yn}] has n+1 sector angles proportional to y0,y1,…,yn *)
Manipulate[
 Rotate[
  PieChart[ {1, a, 4, 2 * a, 4, 3 * a, 1},
   ColorFunction → "Rainbow", ImageSize → Tiny ],
  Pi / 2],
 {a, 0, 5}]
```



Controls

Manipulate lets you set up any number of controls.
You just give the information for each variable in turn, one after another.

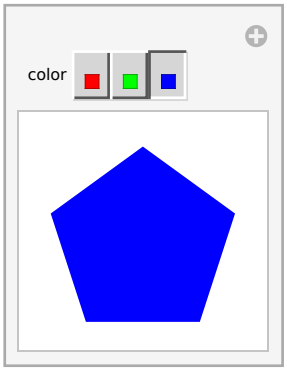
```
Manipulate[
 Graphics[
  Style[ RegularPolygon[n], Hue[h] ], ImageSize → Tiny],
 (* n. of sides in the Regular Polygon *)
 {{n, 6, "n.sides"}, 3, 12, 1},
 (* color *)
 {{h, 0.5, "hue"}, 0, 1}
]
```



There are many ways to specify **Controls** for **Manipulate.**

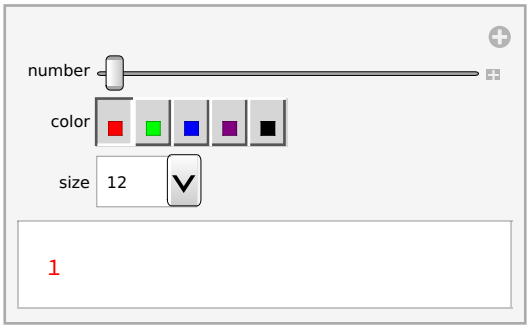
- A list of values (up to 5 values in total) generates a **chooser**

```
(* Here, the  Chooser  renders  a pentagon  in 1 color  choosen among  3 *)
Manipulate[
 Graphics[ Style[ RegularPolygon[5], color ],
           ImageSize → Tiny ],
 {color, {Red, Green, Blue}}
]
```



- If there are more than **5** choices, **Manipulate** sets up a **drop-down menu**:

```
(* Here, we choose a number  with a slider,
we choose 1 out of  5 colors with a chooser,
we choose 1 out of 7 font  sizes with a drop-down menu == menu a tendina *)
Manipulate[
 Style[ number , color, size ],
 {number, 1, 20, 1}, (* slider *)
 {color, { Red, Green, Blue, Purple, Black }}, (* chooser *)
 {size, Range[12, 84, 12]} (* drop-down menu *)
]
```



More Examples

Example of Manipulate with **Nest** (and with a Pure Function) .

We also use a built-in symbol (Subsuperscript) that has no built-in meanings,
i.e. an object that for which no Evaluation Rules are initially defined;
even so, it can be used to construct and expression.

Lets see this with Superscript.
Superscript[x,a] is an object that formats as x^a (2-dimensional formatting).
But it is not Power[x,a].
Superscript is useful, for example, to define $x^{\ddagger}$

```
(* Superscript[x,a]  is  an object that formats as x^a *)
ss = Superscript[x, ‡]
```

$x^{\ddagger}$

```
(* Superscript[]  and Power[] are not the same operator *)
pow = x^‡;
ss === pow
```

```
False
```

```
Map[FullForm, {ss, pow}]
```

```
{Superscript[x, \[DoubleDagger]], Power[x, \[DoubleDagger]]}
```

```
(* NOTE.  Palette  creates  ■□  as  a  Power  *)
palette = x‡ ;
palette == pow
```

```
True
```

In the following Example of Manipulate with **Nest** (and with a Pure Function), we use Subsuperscript .

```
(* Subsuperscript  is  a  Built-in *)
{sss = Subsuperscript[x, p, a], Subsuperscript[x, x, x]}
```

```
{x_p^a, x_x^x}
```

```
myF[r_] := Subsuperscript[r, r, r];
(* myF[r_,s_,t_]:= Subsuperscript[r,s,t]; *)
myF[x]
```

```
x_x^x
```

```
Nest[ myF, x, 3 ]
(*  Nest applies myF to "x"  for  3  times :
  {x,   myF[x],   myF[myF[x]],   myF[myF[myF[x]]]}
   and yields the Last result *)
```

```
x_(x x_x^x)^(x_(x x_x^x)^x)
```

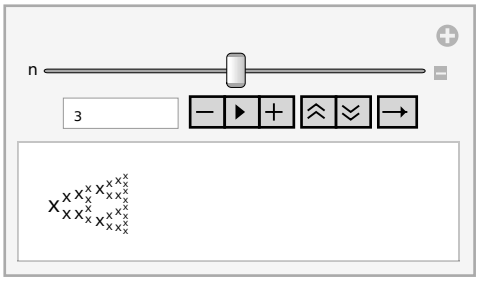
```
NestList[myF, x, 3]
(*  NestList gives the list of applications of myF  to  "x" *)
```

```
{x, x_x^x, x_(x x_x^x)^x, x_(x x_x^x)^(x_(x x_x^x)^x)}
```

```
(*  Nest[ f, expr, n] gives, as output, f applied  n  times to  expr  *)
(*  For example :  *)
(*  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"  for   n  times  *)
 {n, 1, 5, 1}]
```


```
(* Version with a Pure Function *)
(*  Subsuperscript[#,#,#]&  *)
(* instead of myF[r_]:= Subsuperscript[r,r,r] *)
(*  *)
(*  Subsuperscript[#1, #2, #3] &  *)
(* instead of  myF[r_,s_,t_]:= Subsuperscript[r,s,t] *)
Manipulate[
 (*  Nest applies Subsuperscript[#,#,#]&  to  "x"  for  n  times  *)
 Nest[ Subsuperscript[#, #, #] &,  x,  n],
 {n, 1, 5, 1}]
```



NOTE. **Catch/Throw**

Nest gives the last element of **NestList**

```
(* Here, we create :
  {(2^2), ((2^2)^2), ((2^2)^2)^2, ..... } *)
NestList[ #^2 &, 2, 5]
Nest[ #^2 &,  2,  5]
```

{2, 4, 16, 256, 65 536, 4 294 967 296}

4 294 967 296

You can use **Catch/Throw** to exit from **Nest** before it is finished

```
(* Condition   *)
(* Catch the first value > 100, and  Throw it *)
Catch[
 Nest[
   If[ # > 10^2,  Throw[#], #^2 ]&,  (* If, Then, Else *)
  2,
  5]
]
```

```
256
```

- **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} ];
```

```
Before Catch/Throw : {a, b, c, d, e} = {a, b, c, d, e}
```

```
3
```

```
After Catch/Throw  : {a, b, c, d, e} = {1, 2, 3, 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} ];
```

```
Before Catch/Throw : {a, b, c, d, e} = {a, b, c, d, e}
```

```
3
```

```
After Catch/Throw  : {a, b, c, d, e} = {1, 2, 3, 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 }
]
```

```
{b, 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 }
]
```

```
d
```

```
(* 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 }
]
```

```
d
```

■ Use of 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] ] (* x=11 ⟹ "overflow"  *)
Catch[ testF[2] + testF[8] ]  (* 2!+8!*)
```

```
overflow
```

```
40 322
```

■ Use of Throw/Catch.

Exit a loop when a criterion is satisfied

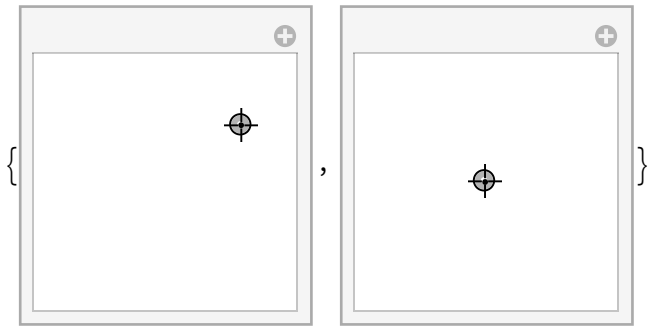
```
(*  It is   13!<10^10  and  14!>10^10 , thus  14 is returned  *)
Catch[
 Do[
  If[ n! > 10^10,  Throw[n] ],
  {n, 20}]
]
```

```
14
```

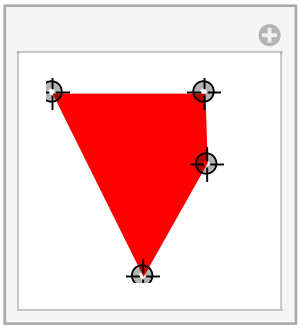
Locator

Locator can be used to control a parameter in a **Manipulate,**
indicating that the value of such parameter is, indeed, given by the position of a locator.

```
{
 Manipulate[
  Graphics[ Point[{p}] , PlotRange → 2, ImageSize → Tiny],
  {{p, {0, 0}}, Locator}],

 Manipulate[
  Graphics[ Point[{p}] , ImageSize → Tiny],
  {{p, {0, 0}}, Locator}]
}
(* PlotRange → 2  is equivalent to PlotRange→{{-2,2},{-2,2}} *)
(*  When you use Graphics inside Manipulate,  set an explicit PlotRange *)
(*  Otherwise, Automatic PlotRange determination will cause Point[{p}] to
 appear not to move,since the PlotRange is always centered around it *)
```

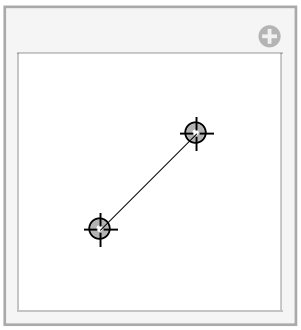


```
Manipulate[
 Graphics[ {Red, Polygon[pt]},  PlotRange → 2, ImageSize → Tiny ],
 {{pt, {{0, 0}, {1, 0}, {1, 1}, {0, 1}}} (* initial vertexes for polygon pt *), Locator}]
```



■ Option LocatorAutoCreate → True
(under Windows/Linux and Mac, not in Cloud)
Add locators with Ctrl Alt (clik) ; su Mac: Cmd (click)
Delete locators with Ctrl Alt (clik) ; su Mac: Cmd (click)

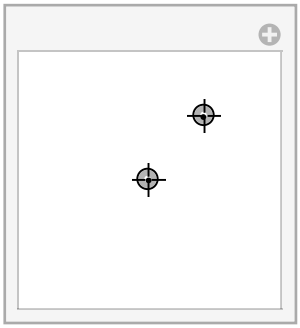
```
Manipulate[
 Graphics[ {Line[u]},  PlotRange → 2, ImageSize → Tiny ],
 {{u, {{-1, -1}, {1, 1}}},
  Locator,
  LocatorAutoCreate → True}]
```



(* Ctrl Shift E per aprire e richiudere una cella e vederne il contenuto *)

■ Option LocatorAutoCreate → All
Add locators with with **any** mouse click
Delete locators with Ctrl Alt (clik) ; su Mac: Cmd (click)

```
Manipulate[
 Graphics[ Point[locs], PlotRange → 2, ImageSize → Tiny],
 {{locs, {{0, 0}}},
  Locator,
  LocatorAutoCreate → All}]
```

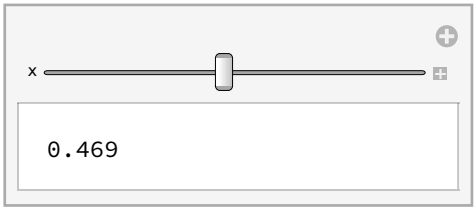


NOTE . SaveDefinitions

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.

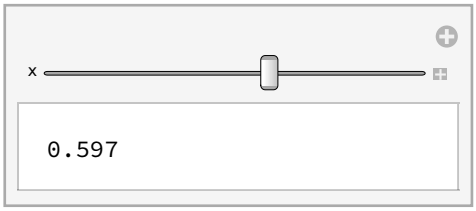
```
f[x_] := x;
```

```
mf = Manipulate[f[x], {x, 0, 1}]
```




```
g[x_] := x;
```

```
mg = Manipulate[g[x], {x, 0, 1}, SaveDefinitions → True]
```



```
? f
```

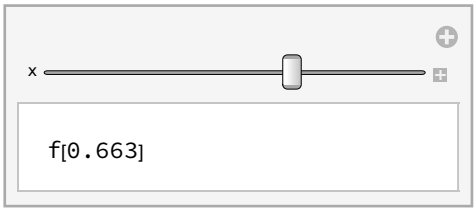
```
Symbol

Global`f

Definitions
f[x_] := x

Full Name Global`f
```

```
? g
```

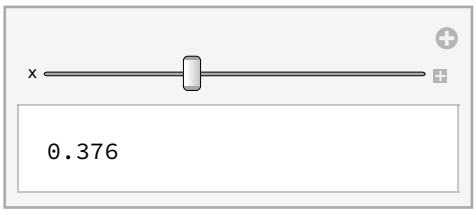
```
Symbol

Global`g

Definitions
g[x_] := x

Full Name Global`g
```

Now, **Quit** the Kernel; then, open a notebook (this one or a new one), evaluate the following statements and watch the different behaviour:

```
Manipulate[f[x], {x, 0, 1}]
```



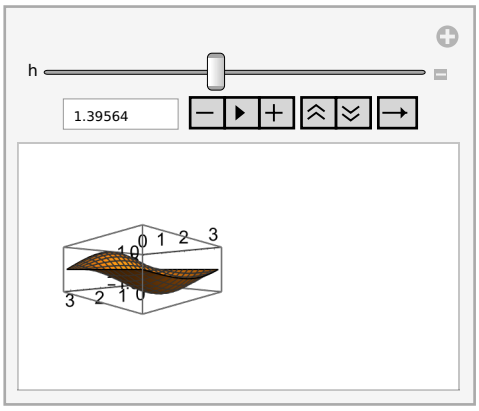

```
Manipulate[g[x], {x, 0, 1}]
```



(* Warning. Saved variable definitions are effectively treated as Global *)

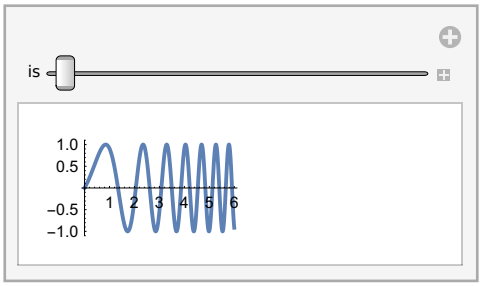
Further Examples

```
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}]  ; *)
```

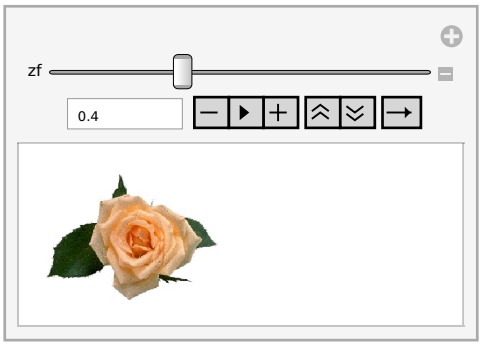
```
Manipulate[
 Plot[Sin[x (1 + x)], {x, 0, 6}, ImageSize → is],
 {is, 100, 200, 10}]
```



Import , Export, SetDirectory, NotebookDirectory

```
rose = Import["ExampleData/rose.gif"];
```

```
Manipulate[Magnify[rose, zf], {zf, 0.1, 1, 0.1}]
```




```
NotebookDirectory[];

SetDirectory[NotebookDirectory[]];

Export["myrose.jpg", rose];

(* Acquire an Image with your PC camera *)
img = CurrentImage[];

Export["myimg.jpg", img];
```

DynamicImage (not nec)

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, NestList
Throw , Catch
Subsuperscript
SetDirectory[], NotebookDirectory[]
Import, Export, CurrentImage
DynamicImage

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.

9.5/9.5bis Make a Manipulate to show a Disk with Red, Green, Blue color components, **each** varying from 0 to 1 (AutoRun; Hue, Setter Bar control)

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. (try using Map & Range)

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

■ **tutorial/IntroductionToManipulate**
https://reference.wolfram.com/language/tutorial/IntroductionToManipulate.html

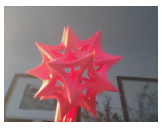
■ The Wolfram Demonstrations Project: more than 11000 interactive Demonstrations created with Manipulate.

■ **tutorial/IntroductionToControlObjects**
https://reference.wolfram.com/language/tutorial/IntroductionToControlObjects.html

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}]
```

```
{512, 512}
```

ExampleData

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;
GraphicsRow[{
   imgDodecaedro ,
   ImageResize[ imgDodecaedro , {width, height}],
   ImageResize[ColorNegate[imgDodecaedro] , {width, height}]
 }, ImageSize → Small]
```

Blur

The second argument n in Blur is a number to specify the range of pixels that get blurred together.

```
Blur[imgDodecaedro] == Blur[imgDodecaedro, 2](* 2 is the default *)
GraphicsRow[{Blur[imgDodecaedro] , Blur[imgDodecaedro, 10] }, ImageSize → Small]
```

```
True
```

```
(* Row puo' essere usato con qualsiasi lista di espressioni *)
Row[{Blur[imgDodecaedro] , Blur[imgDodecaedro, 10]}]
(* Il terzo argomento "  " inserisce uno spazio tra le immagini *)
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:

```
(* Table[n, {n,0,15,5}] gives {0,5,10,15} *)
ImageCollage[
 Table[
  Blur[imgDodecaedro, n],
  {n, 0, 15, 5}],
 ImageSize → Tiny]
```

Create a collage from a weighted list of images

```
{b0, b5, b15, b40} = Map[Blur[imgDodecaedro, #] &, {0, 5, 15, 40}];
(* b0=Blur[imgDodecaedro,0];
b5=Blur[imgDodecaedro,5];
b15=Blur[imgDodecaedro,15];
b40=Blur[imgDodecaedro,40]; *)
Row[{b0, b5, b15, b40}]
(* ImageCollage[ { w1→img1, w2→img2, ... } ]
 creates a collage of images  img_i
 based on their corresponding weights  w_i *)
ImageCollage[{
  1 → b0,
  4 → b5,
  9 → b15,
  1 → b40},
 ImageSize → Small]
```

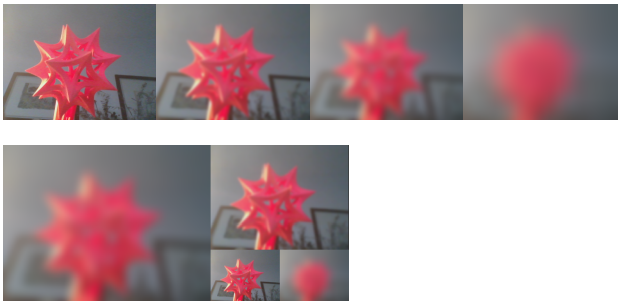


Image Analysis

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 in 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

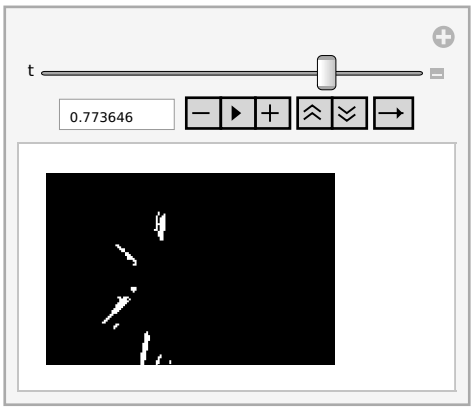
Another type of analysis is **edge detection** :
finding where in the image there are **sharp changes** in color.

```
width = 150;
height = 100;
imgDodecaedroEdge = EdgeDetect[imgDodecaedro];
ImageResize[imgDodecaedroEdge, {width, height}]
```

```
(* Add the 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 -- example of Optional Argument)

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 :

```
? WikipediaData
```

```
wdList = WikipediaData["Mare Tranquillitatis", "ImageList"];
wd = wdList[[1]];
ImageResize[wd, 200]
(* wdListShort=Flatten[{wdList[[1;;4]],wdList[[7]],wdList[[9;;13]],wdList[[16;;17]]}]; *)
```

- 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.

```
Row[{
  imgDodecaedro,
  Sqrt[imgDodecaedro],
  imgDodecaedro - EdgeDetect[imgDodecaedro]
 }]
```

```
draw = ColorNegate[EdgeDetect[imgDodecaedro]] + imgDodecaedro ;
ImageResize[draw, 250]
```