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}
```

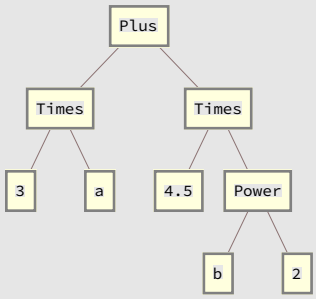
```
Power^2[4, 2] === Power[4, 2]^2
(* NOTA: Se si vuole eseguire la comparazione con Equal, invece che con SameQ,
e si vuole ottenere sempre un Booleano, si puo' usare  TrueQ *)
 TrueQ[Power^2[4, 2] == Power[4, 2]^2]
```

```
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
```

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

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]}}
```

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

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).

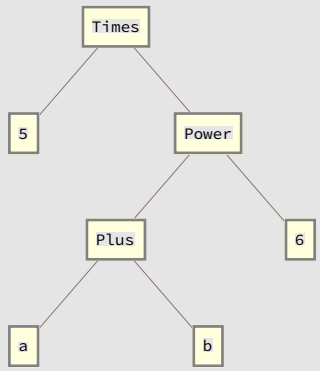
```
lista = {a, a + b, a + a};
(* Cases  estrae  a+b  da  lista *)
Cases[lista, x_ + y_]
(* Cases  estrae  a+b  da  lista  e  lo  sostituisce  con  b *)
Cases[lista, x_ + y_ → y]
```

```
{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.

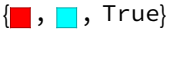
You can refer to common colors by their names.

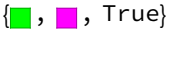
```
{Red, Green, Blue, Purple, Orange, Black}
(* swatch *)
```

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 }
```





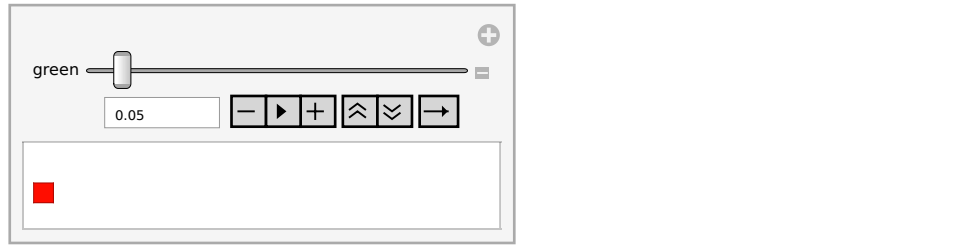
▪ **Blend** blends a list of colors together.

```
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 ]}
```

{■, ■, ■, ■}

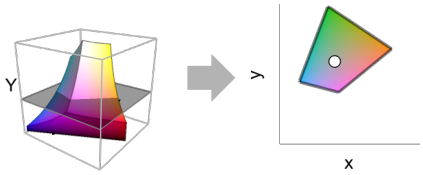
You can specify named HTML colors (e.g. RGBColor["aqua"]) as well as hex colors (e.g. RGBColor["#00ff00"])

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

{■, True, ■, True}

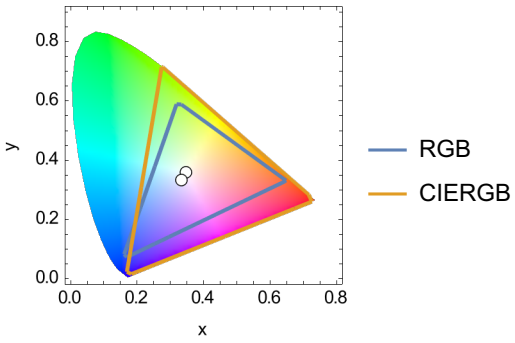
ChromaticityPlot and **ChromaticityPlot3D** plot lists of colors in color space.

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).

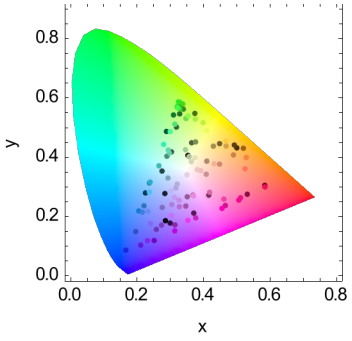


→ It can be used to compare to color space models.

```
(* 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**.

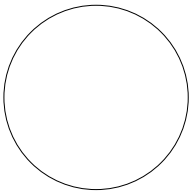
Basic Graphics Objects

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]
```

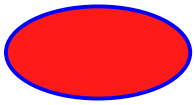
Use the graphics transparency directive **Opacity :**

```
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] *)
```

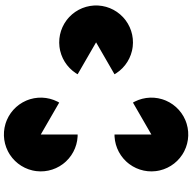
Use the graphics directive **EdgeForm :**

```
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 *)
```

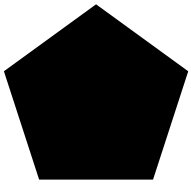
```
pie[data]
```

```
{{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]
```



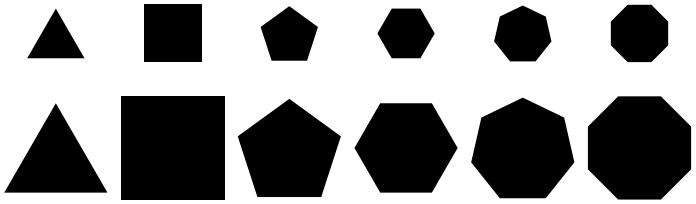
You can find a **list** of REGULAR POLYGONS NAMES , for instance, at
https : // www.mathsisfun.com/geometry/polygons.html

```
(*  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 ];
```

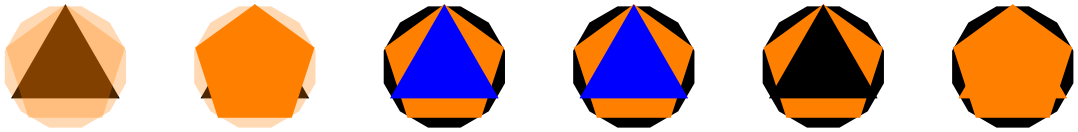
Style works inside **Graphics**, so you can use it to give colors.

```
(*
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];
```

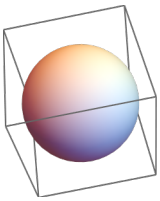
```
(*  Flatten, di default, appiattisce tutti i livelli,
restituendo una lista mono-dimensionale *)
p4 = Flatten[p3];
g4 = Graphics[p4];
(* Flatten mostra che e' superfluo, in p3, usare List attorno a { Orange, penta } *)
FullForm[p3];
FullForm[p4];

(* 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**.

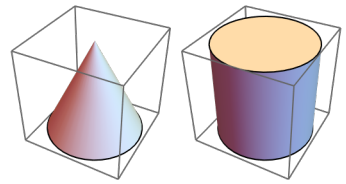
You can rotate 3D graphics interactively to see different angles.

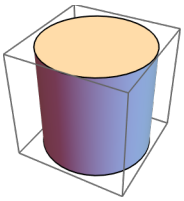
```
Graphics3D[
  Sphere[],
 ImageSize → Tiny]
```

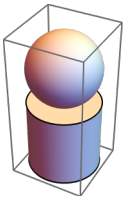


- A **list** of Graphics3D directives (Cone and Cylinder).
- One Graphics3D, whose argument is a list of graphics objects (Sphere and Cylinder) .

```
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]
```

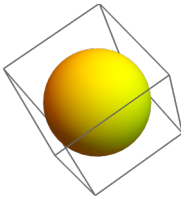






A yellow sphere; note that it is rendered like an actual 3D object, with lighting;
if it were pure yellow, we would not see any 3D depth, and it would look like a 2D disk.

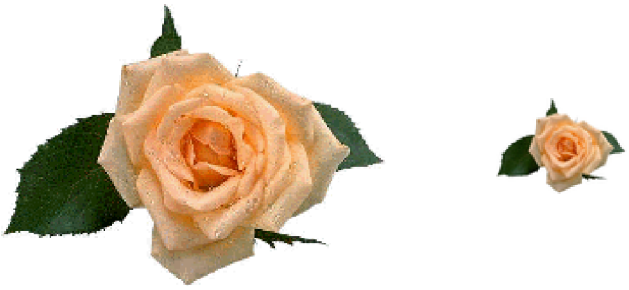
```
Graphics3D[
 Style[Sphere[], Yellow],
 ImageSize → Tiny]
```



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)
```

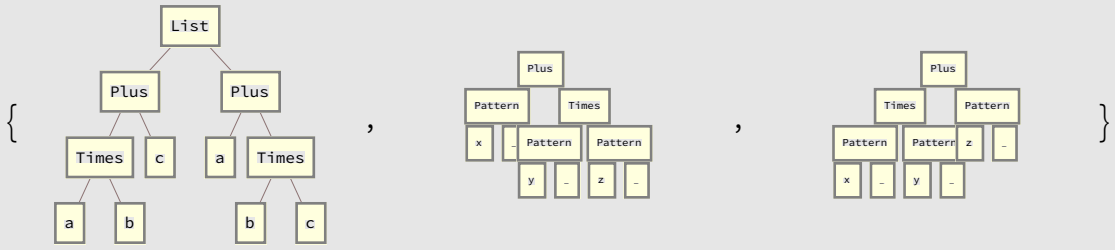
```
True
```

```
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]
```

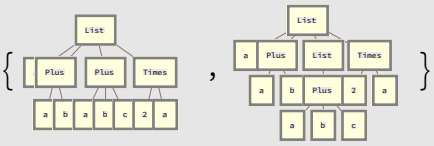
```
{a + b, a + b + c}
```

```
2
```

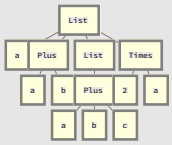
```
{{2}, {3}}
```

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]
```



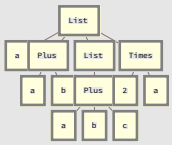

```
{a + b}
```

```
{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]
```



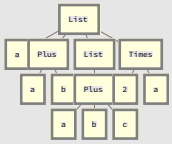
1

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];
```



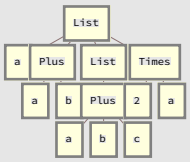
{{2}}

{{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]
```

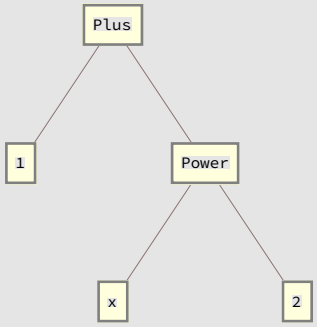
```
log[a] + log[b c d]
```

```
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.

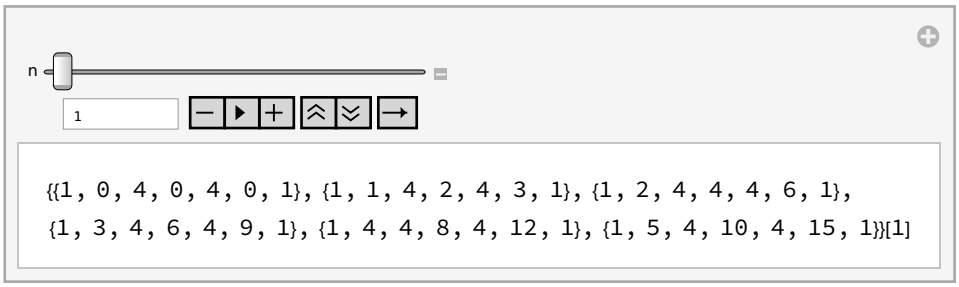
The built-in **Manipulate** serves to create an interface where we can continually manipulate one or more variables (parameters).

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}]
```

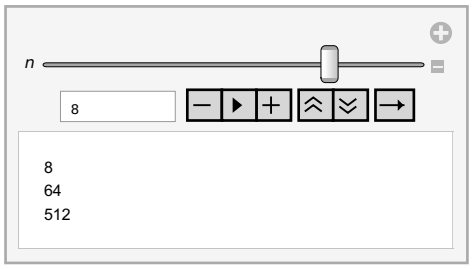
{{■}, {■, ■}, {■, ■, ■}, {■, ■, ■, ■}, {■, ■, ■, ■, ■}}

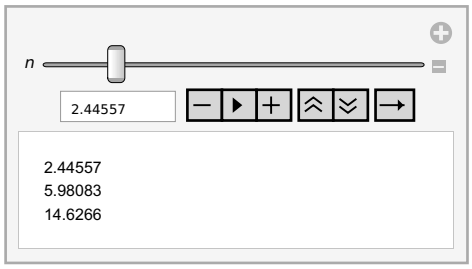


By default, Table assumes **start=step=1**.
Manipulate needs **start** to be specified.
If you omit the step size, Manipulate uses a non-integer step
(it assumes you want machine-precision values, in the specified range).

```
(* 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
```



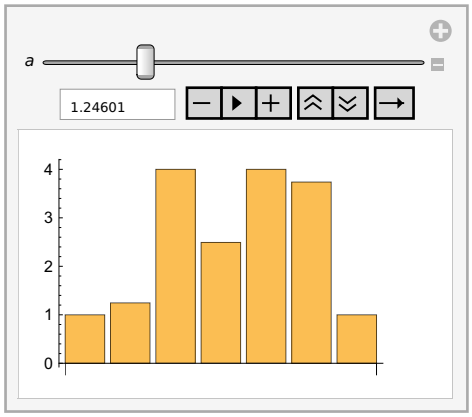


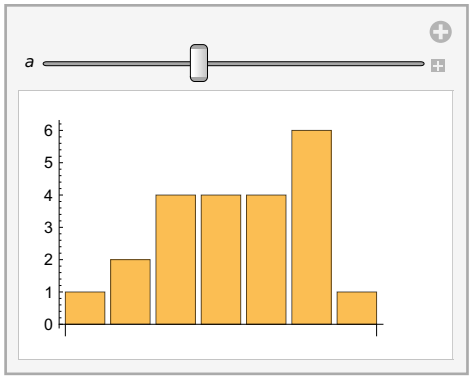

```
ff[n_] := Column[Map[FullForm, {n, n^2, n^3}]];
Manipulate[ff[n], {n, 1, 10}];
```

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}]
 }]
```



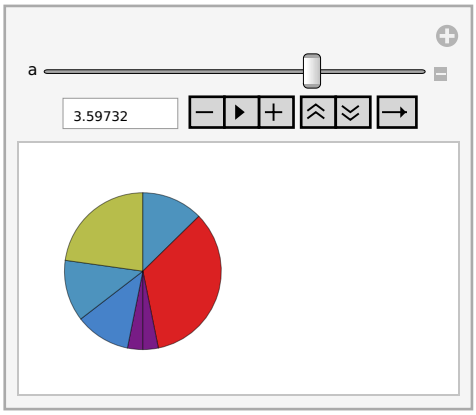


A pie chart that changes as you move the slider :

```
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}}
```

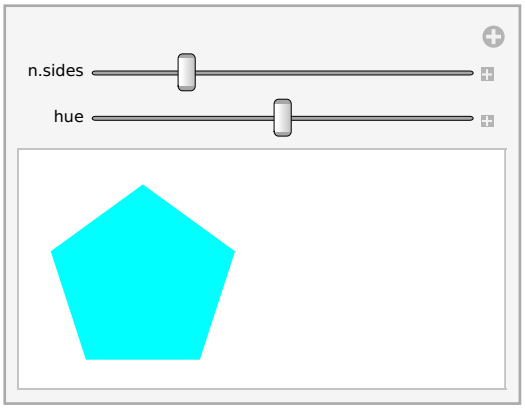
```
(* PieChart[{y0,y1,…,yn}] has n+1 sector angles proportional to y0,y1,…,yn *)
Manipulate[
 Rotate[
  PieChart[ {1, a, 4, 2 * a, 4, 3 * a, 1},
   ColorFunction → "Rainbow", ImageSize → Tiny ],
  Pi / 2],
 {a, 0, 5}]
```



Controls

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

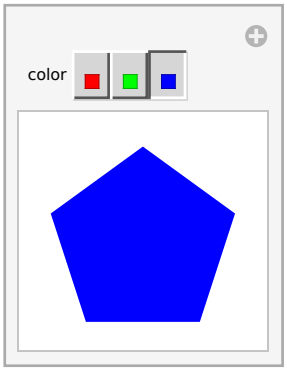
```
Manipulate[
 Graphics[
  Style[ RegularPolygon[n], Hue[h] ], ImageSize → Tiny],
 (* n. of sides in the Regular Polygon *)
 {{n, 6, "n.sides"}, 3, 12, 1},
 (* color *)
 {{h, 0.5, "hue"}, 0, 1}
]
```



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

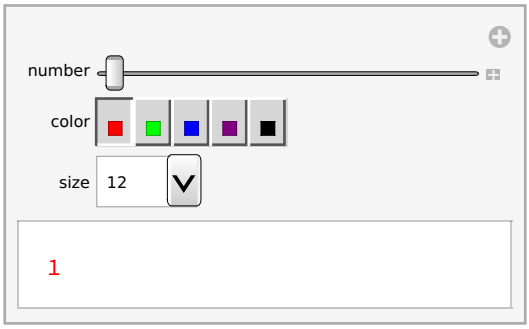
- A list of values (up to 5 values in total) generates a **chooser**

```
(* 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)
```

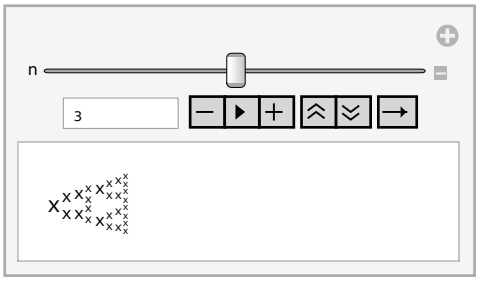
```
NestList[myF, x, 3]
(*  NestList gives the list of applications of myF  to  "x" *)
```

```
{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

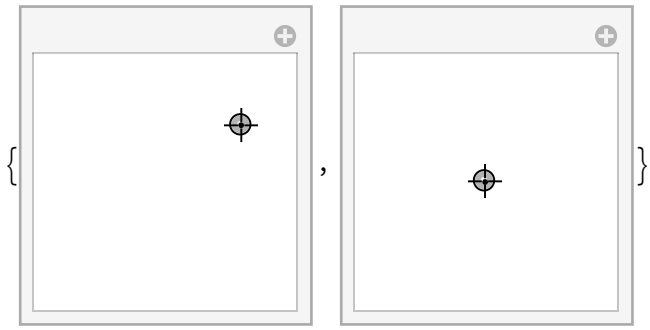
```
(*  It is   13!<10^10  and  14!>10^10 , thus  14 is returned  *)
Catch[
 Do[
  If[ n! > 10^10,  Throw[n] ],
  {n, 20}]
]
```

```
14
```

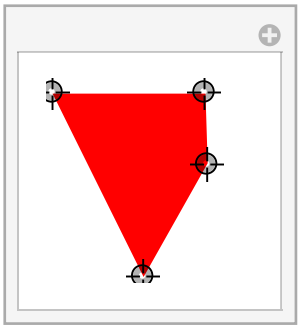
Locator

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 *)
```

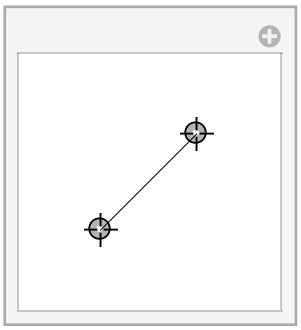


```
Manipulate[
 Graphics[ {Red, Polygon[pt]},  PlotRange → 2, ImageSize → Tiny ],
 {{pt, {{0, 0}, {1, 0}, {1, 1}, {0, 1}}} (* initial vertexes for polygon pt *), Locator}]
```



■ 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)

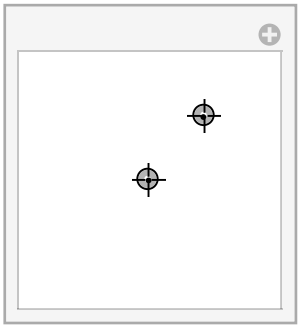
```
Manipulate[
 Graphics[ {Line[u]},  PlotRange → 2, ImageSize → Tiny ],
 {{u, {{-1, -1}, {1, 1}}},
  Locator,
  LocatorAutoCreate → True}]
```



(* 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}]
```

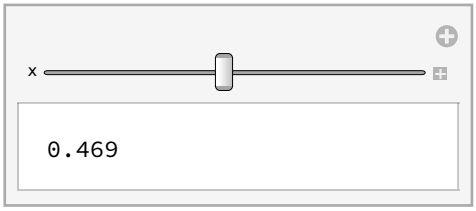


NOTE . SaveDefinitions

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.

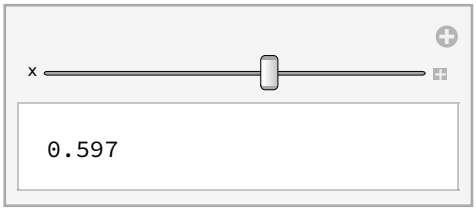
```
f[x_] := x;
```

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




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

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



```
? f
```

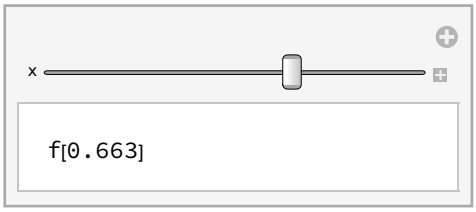
```
Symbol

Global`f

Definitions
f[x_] := x

Full Name Global`f
```

```
? g
```

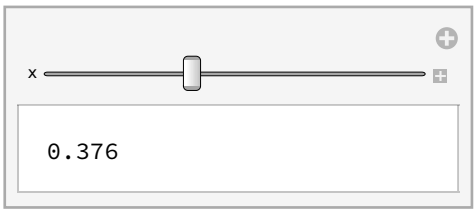
```
Symbol

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}]
```



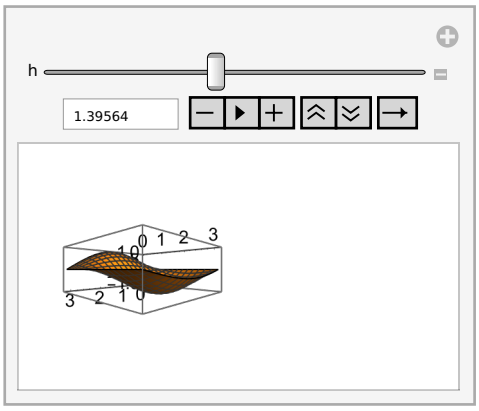

```
Manipulate[g[x], {x, 0, 1}]
```



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

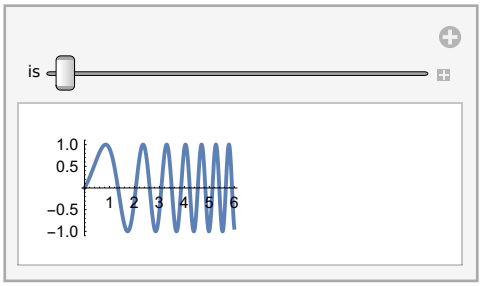
Further Examples

```
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}]  ; *)
```

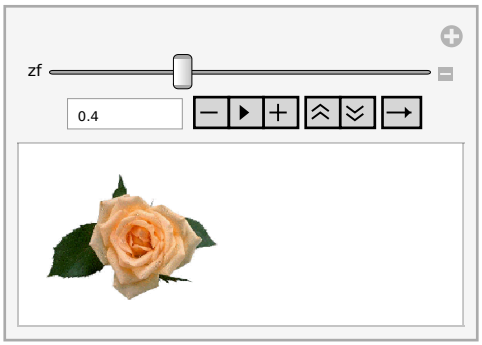
```
Manipulate[
 Plot[Sin[x (1 + x)], {x, 0, 6}, ImageSize → is],
 {is, 100, 200, 10}]
```



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

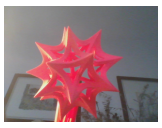
■ The Wolfram Demonstrations Project: more than 11000 interactive Demonstrations created with Manipulate.

■ **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]
```

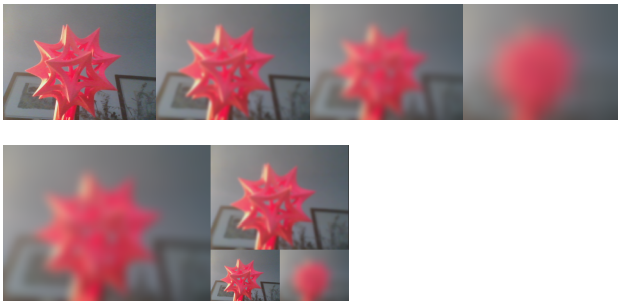


Image Analysis

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

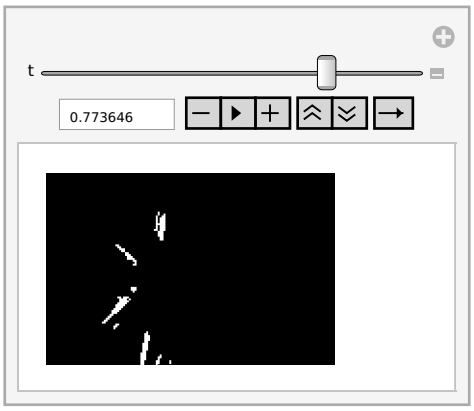
Another type of analysis is **edge detection** :
finding where in the image there are **sharp changes** in color.

```
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
```

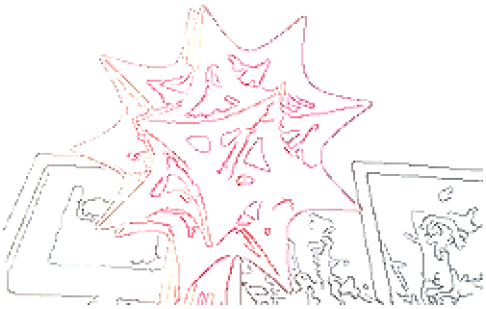
```
wdList = WikipediaData["Mare Tranquillitatis", "ImageList"];
wd = wdList[[1]];
ImageResize[wd, 200]
(* wdListShort=Flatten[{wdList[[1;;4]],wdList[[7]],wdList[[9;;13]],wdList[[16;;17]]}]; *)
```

- 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]
```



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]
```

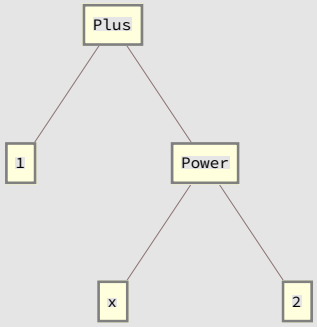
```
log[a] + log[b c d]
```

```
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}}}

Module e Block

Uno dei modi per essere sicuri che parti differenti di un programma non interferiscano è quello di dare alle rispettive variabili una determinata ‘‘visibilità’’ (scope).

Nei linguaggi dotati del concetto di blocco (come C, C++, Pascal, Java e molti altri), una variabile visibile all’interno di un blocco è in generale visibile anche all’interno di eventuali blocchi annidati.

Le regole fondamentali di visibilità sono in genere modificate dalla regola speciale dello “shadowing”, secondo cui una variabile locale “nasconde” una eventuale variabile omonima definita nello scope superiore.

In altre parole, se in un programma è definita una variabile globale e in un determinato sottoprogramma viene definita una variabile locale omonima, il sottoprogramma in questione perde la visibilità della variabile globale, nascosta da quella locale.

Module corrisponde a **scoping statico** o **lessicale** (localizza i nomi di variabile): le variabili sono trattate come locali ad una determinata sezione del codice (in un programma).

Block corrisponde a **scoping dinamico** (localizza i valori delle variabili): le variabili sono trattate come locali ad una determinata parte della history di esecuzione (di un programma).

- **Esempio**
  - **Block**

```
(*  Considero  z  e  x variabili globali   *)
(*  Non assegno valori alla z nel contesto globale  *)
(*  Assegno un valore ad x nel contesto globale : x=1  *)
Clear[z, x];   z;   x = 1;
Print["Prima di Block:  x = ", x, ",   z = ", z];

Block[ {x},  (*  x è dichiarata locale  (x verde)
⟹ il simbolo x locale ha lo stesso nome del simbolo x globale,
              ma  i  loro  contenuti  sono  diversi
             (il contenuto di x globale è salvato a parte)
              Non assegno valori a  x  dentro Block. *)
  z = x;   (*  z   non è dichiarata locale
            ⟹ z rimane nota a livello globale  (z nera, dopo questa Set)
            ⟹ la sua assegnazione z=x è nota globalmente,
               anche se avviene dentro Block  *)
  Print["Dentro Block:    x = ", x, ",   z = ", z];
 ];

Print["Finita  Block:   x = ", x, ",   z = ", z];
(* x torna simbolo globale
⟹ il contenuto di x torna quello che era prima di Block (x=1)  *)

(* z  è rimasta nota a livello globale anche dentro Block
⟹ z  non è più non-assegnata (come era prima di Block):
   z  ha il contenuto (z=x) noto globalmente
      (anche se assegnato dentro Block) *)

x = 2;   (* Se ri-assegno x , ri-assegno anche  z *)
Print["Ri-assegnata x:  x = ", x, ",   z = ", z];
```

```
Prima di Block:  x = 1,   z = z
Dentro Block:    x = x,   z = x
Finita  Block:   x = 1,   z = 1
Ri-assegnata x:  x = 2,   z = 2
```

■ Module

```
(*  Considero  z  e  x variabili globali   *)
(*  Non assegno valori a z nel contesto globale  *)
(*  Assegno un valore a x nel contesto globale : x=1  *)
Clear[z, x];    x = 1;
Print["Prima di Module: x = ", x, ",        z = ", z];

Module[ {x}, (* x è dichiarata locale   (x verde)
 ⟹ viene creato  x$num  locale, diverso da  x  globale.
    Non assegno valori a  x$num  dentro Module *)
  z = x;   (* z   non è dichiarata locale
           ⟹ z rimane nota a livello globale (z nera, dopo questa Set)
           ⟹ la sua assegnazione z=x$num  è nota globalmente,
                anche se avviene dentro Module *)
  Print["Dentro Module:   x = ", x, ",   z = ", z];
 ];

Print["Finita  Module:  x = ", x, ",        z = ", z];
(*  x è (sempre rimasto) simbolo globale, col suo contenuto *)

(*  z  è rimasta nota a livello globale anche dentro Module
⟹ z  non è più non-assegnata (come era prima di Module):
   z  ha il contenuto (z=x$num) noto globalmente
      (anche se assegnato dentro Module) *)

x = 2;  (* Se ri-assegno x , non ri-assegno z  *)
Print["Ri-assegnata x:  x = ", x, ",        z = ", z];
```

```
Prima di Module: x = 1,        z = z
Dentro Module:   x = x$3723,   z = x$3723
Finita  Module:  x = 1,        z = x$3723
Ri-assegnata x:  x = 2,        z = x$3723
```

Riporto qui gli output ottenuti con Block, per comodita' di paragone

```
Prima di Block:  x = 1,   z = z
Dentro Block:    x = x,   z = x
Finita  Block:   x = 1,   z = 1
Ri-assegnata x:  x = 2,   z = 2
```

Come funziona Module

Ogni volta che Module è usata, viene creato un nuovo simbolo per rappresentare ciascuna delle sue variabili locali.

A tale nuovo simbolo viene dato un nome univoco, che non può entrare in conflitto con nessun altro nome.
Tale nome univoco è formato dal nome della variabile locale stessa, seguito da $num (in cui num è un numero seriale, che viene progressivamente incrementato).

```
Clear[t, tglobal];
t = tglobal;
Print["t prima di Module:  ", t];

Module[{t},
  Print["t dentro  Module:   ", t];
  t];

Print["t dopo Module:  ", t];
```

```
t prima di Module:  tglobal
t dentro  Module:   t$3725
t dopo Module:  tglobal
```

Posso usare il simbolo **t** , creato e restituito da Module, anche fuori da essa:

```
Clear[t];
outm = Module[{t}, t];
1 + outm^2
```

$$1 + t\$3744^2$$

Esempio 2 (non nec)

Esempio 3 (non nec)

Come funziona Block

Block crea una parte di codice (un environment) in cui è possibile cambiare temporaneamente il **valore** delle variabili di interesse.
Espressioni, che vengono valutate durante l'esecuzione di una Block, usano i valori definiti correntemente (localmente nella Block).
Questo è vero sia nel caso in cui tali espressioni appaiano direttamente (come parti di un body di Block), sia nel caso in cui tali espressioni vengano prodotte dalla valutazione stessa.

```
Clear[t, tglobal];
t = tglobal;
Print["t prima di Block:  ", t];

Block[{t},
  Print["t dentro  Block:   ", t];
  t];
(*  Anche  se t è stata assegnato  globalmente al valore tglobal,
qui ho dichiarato t locale  a  Block  e  non  gli  ho dato  alcun  valore
 ==> quindi  Print[t]  restituisce il simbolo  t
  che  è  il  valore  localmente dato  a  t dentro Block .

 ==> Block  restituisce  il  simbolo  t ,
il quale (appena  Block termina) viene  valutato  e  riconosciuto
 come  simbolo  globale ,  cui  è  assegnato  il  valore  tglobal  *)
Print["t dopo Block:  ", t];
```

```
t prima di Block:  tglobal
t dentro  Block:   t
t dopo Block:  tglobal
```

(* Riporto gli output ottenuti con Module, per comodita' di paragone *)

```
t prima di Module:  tglobal
t dentro  Module:   t$4661
t dopo Module:  tglobal
```

Posso usare il simbolo **t** (dichiarato locale a Block), anche fuori da essa:

```
Clear[t];
outb = Block[{t}, t];
1 + outb ^ 2
```

$1 + t^2$

Esempio 2 (non nec)

Esempio 3 (non nec)

- **Quando è utile Block?**
  - **In costrutti iterativi (e.g. Do, Sum, Table)**
  - **Block, per lavorare in Precision pre - fissata**

Strings and Text

Mathematica allows computing with text.
You enter text as a string, indicated by quotes (“ ... ”) ,
A string is an atom.

There are 3 types of **atom** : symbol, number, string (of characters).
▪ **Symbol**: sequence of letters and/or numbers and/or the character $; a symbol cannot start with a number.
▪ There are 4 types of numbers.
▪ **Integer** , exact (a sequence of decimal digits *dddddd);*
▪ **Rational** , exact (it has the form Integer1/Integer2, reduced to lowest terms);
▪ **Real**, approximate, with any specified precision (it has the form *ddd.ddd* i.e. it is a variable-precision floating-point);
▪ **Complex** (it has the form a+b*I , where a, b can be any of the previous number types).
▪ **String** of characters: any sequence of characters in between open/closed double quotes “ ... ”

Just like when you enter a number, a string (on its own) comes back unchanged, except that the quotes are not visible, when the string is displayed in output.

```
"This is a string."
AtomQ[%]
```

```
This is a string.
```

```
True
```

There are many functions that work on strings.

StringLength, StringReverse

StringLength gives the length of a string, by counting the number of its characters:

```
StringLength["Hello World"]
StringLength["Hello    World"]
StringLength[{"Hello World", "Hello    World"}]
```

```
11
```

```
14
```

```
{11, 14}
```

StringReverse reverses the characters in a string:

```
(* StringReverse["Hello World"]; *)
(* StringReverse["If  I  had  a  Hi-Fi"]  ; *)
StringReverse["Don't nod"]
```

```
don t'noD
```

ToUpperCase

ToUpperCase makes all the characters in a string uppercase (capital letters):

```
ToUpperCase["Hello World"]
```

```
HELLO WORLD
```

StringTake, StringJoin

StringTake takes a certain number of characters from a string.
If you take **n** characters, you get a **string** of length **n** .

```
st = StringTake["Hello World", -5]
StringLength[st]
```

```
World
```

```
5
```

StringJoin joins strings (do not forget spaces, if you want separate words):

```
StringJoin["Hello", "World"]
StringJoin["Hello", " World"]
```

```
HelloWorld
```

```
Hello World
```

You can do “computations” with strings.

```
stringList = {"apple", "banana", "strawberry"};
(* get characters for each component in the list *)
StringJoin[
 StringTake[stringList[[1]], -3],
 StringTake[stringList[[2]], -2],
 StringTake[stringList[[3]], -2]]
```

```
plenary
```

Characters breaks a string into a list of its characters

Sometimes it is useful to turn strings into lists of their characters.
Each Character is a string of Length 1.

```
Characters["A string is made of characters"]
```

```
{A,  , s, t, r, i, n, g,  , i, s,  , m, a, d, e,  , o, f,  , c, h, a, r, a, c, t, e, r, s}
```

Once a string is broken into a list of characters, all the usual built-in (for Lists) can be used on it.

```
cc = Characters["An apple or two"];
scc = Sort[cc]
FullForm[scc]
(* Anche gli spazi sono nella List in output *)
(*  TableForm[{
    InputForm[scc],
    FullForm[scc],
    OutputForm[scc]}] ; *)
```

```
{ ,  ,  , a, A, e, l, n, o, o, p, p, r, t, w}
```

```
List[" ", " ", " ", "a", "A", "e", "l", "n", "o", "o", "p", "p", "r", "t", "w"]
```

TextWords, TextSentences

Functions like **StringJoin**, or **Characters**, work on strings of any kind (strings that do not need to be meaningful).
Other functions are useful on meaningful text (written, say, in English).
TextWords , for instance, gives a list of the words in a string of text:

```
oneSentence =
  "This is a sentence: sentences are made of phrases; phrases are made of words.";
tw = TextWords[oneSentence]
(* Length of each word in the list  tw *)StringLength[tw]
```

```
{This, is, a, sentence, sentences, are, made, of, phrases, phrases, are, made, of, words}
```

```
{4, 2, 1, 8, 9, 3, 4, 2, 7, 7, 3, 4, 2, 5}
```

TextSentences breaks a text string into a list of sentences :

```
oneSentence =
 "This is a sentence: sentences are made of phrases; phrases are made of words.";
fewSentences =
 "This is a sentence. Sentences are made of phrases. Phrases are made of words.";
(* A seconda della punteggiatura, ho 1 frase oppure piu' frasi *)
ts0 = TextSentences[oneSentence]
ts = TextSentences[fewSentences]
(* Length of each sentence *)
StringLength[ts0]
StringLength[ts]
```

```
{This is a sentence: sentences are made of phrases; phrases are made of words.}
```

```
{This is a sentence., Sentences are made of phrases., Phrases are made of words.}
```

```
{77}
```

```
{19, 30, 26}
```

WikipediaData , WordCloud

There are many ways to get text into *Mathematica*.
One example is the **WikipediaData** function, which gets the current text of Wikipedia articles.

```
(*  Wikipedia  article on “computers” *)
wdPC = WikipediaData["computers"];

(*  Get the first 120 characters of such article *)
PC113 = StringTake[wdPC, 120]
```

```
A computer is a machine that can be programmed to
  automatically carry out sequences of arithmetic or logical operations
```

```
(* wdTS contains the first  3  sentences of the wdPC article   *)
wdTS = TextSentences[wdPC, 3]
(* 2nd  element of wdTS, ie the 2nd sentence of the wdPC article  *)
Part[wdTS, 2]
```

```
{A computer is a machine that can be programmed to automatically carry
    out sequences of arithmetic or logical operations (computation).,
 Modern digital electronic computers can perform generic
    sets of operations known as programs.,
 These programs enable computers to perform a wide range of tasks.}
```

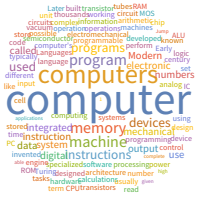
```
Modern digital electronic computers can
  perform generic sets of operations known as programs.
```

```
(* Lets check *)
Hyperlink["https://en.wikipedia.org/wiki/Computer", Appearance → "DialogBox"]
```

https://en.wikipedia.org/wiki/Computer

Given a piece of text, to get a sense of its content, a convenient way is to create a **WordCloud** .

```
WordCloud[wdPC, ImageSize → Tiny]
(* not surprisingly,
"computer" and "computers" are the most common words in the article *)
```



WordList

Mathematica has lots of built-in knowledge about words (in English and other languages).
WordList gives lists of words.

```
(*  Get the first 5 words from a list of common English words *)
wlEnglish = WordList[];
Length[wlEnglish]
Take[wlEnglish, 5]
```

39 176

{a, aah, aardvark, aback, abacus}

Note: aardvark = formichiere
to shift aback = tirare indietro
to take aback = prendere alla sprovvista

```
(*  Get the last 5 words from a list of common Italian words *)
wlIta = WordList[Language → "Italian"];
Length[wlIta]
Take[wlIta, -5]
```

116 854

{zuppe, zuppi, zuppiera, zuppo, zuzzurellone}

Let us create a **WordCloud** from the first letters of all the words

```
(* wlEnglish=WordList[]; *)
wlst1 = StringTake[wlEnglish, 1];
WordCloud[
 wlst1,
 FontFamily → "Arial",
 ImageSize → Tiny]
```

```
Length[$FontFamilies]
```

```
2541
```

```
WordCloud[
  wlst1,
  FontFamily → "Cabin",
  ImageSize → Tiny];
```

```
WordCloud[
  wlst1,
  FontFamily → "Comic Sans MS",
  ImageSize → Tiny];
```

RomanNumeral

Strings do not have to contain text.
We can, for example, generate Roman numerals as strings.

```
RomanNumeral[1987]
Head[%]
FullForm[%%]
```

```
MCMLXXXVII
```

```
String
```

```
"MCMLXXXVII"
```

Make a table of the first 20 Roman numerals .

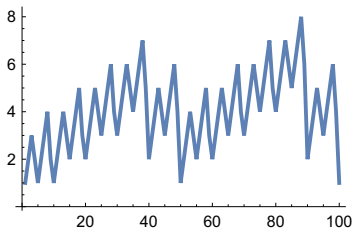
```
Table[ RomanNumeral[n], {n, 20}]
```

```
{I, II, III, IV, V, VI, VII, VIII, IX, X, XI, XII, XIII, XIV, XV, XVI, XVII, XVIII, XIX, XX}
```

As with everything, we can do computations on these strings.
For example, here we plot the **lengths** of successive Roman numerals.

```
rn = Table[ RomanNumeral[n] , {n, 100} ];
tr = StringLength[ rn ];
{Min[tr], Max[tr]}
(* I,V,X,L,C, i.e. 1,5,10,50,100,  have min Lenght  *)
(* LXXXVIII, i.e. 88, has  max Lenght *)
ListLinePlot[tr, ImageSize → Small]
```

```
{1, 8}
```



IntegerName

IntegerName gives the (English) name of an integer

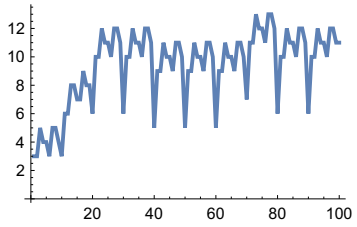
```
{IntegerName[123],
 IntegerName[123, Language → "Italian"]}
```

```
{one hundred twenty-three, centoventitré}
```

Plot the Lengths of the (English) names of the first 100 integers:

```
in = Table[IntegerName[n], {n, 100}];
tin = StringLength[in];
{Min[tin], Max[tin]}
ListLinePlot[tin, ImageSize → Small]
(* {one,two,six,ten}  have min Lenght  *)
(* {seventy-three,seventy-seven,seventy-eight} have  max Lenght *)
```

```
{3, 13}
```



Alphabet, LetterNumber, Transliterate

There are various ways to turn letters into numbers (and vice versa).
Alphabet gives the (English) alphabet:

```
Alphabet[]
(*  Alphabet[Language→ "Italian"]  *)
Alphabet["Italian"]
```

```
{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z}
```

```
{a, b, c, d, e, f, g, h, i, l, m, n, o, p, q, r, s, t, u, v, z}
```

LetterNumber tells you where, in the (English) alphabet, a letter appears.
The opposite is done by **FromLetterNumber**[**n**], that gives the (lowercase) letter at position **n** in the (English) alphabet.

```
LetterNumber[{"a", "b", "x", "y", "z"}]
FromLetterNumber[{1, 2, 24, 25, 26}]
```

```
{1, 2, 24, 25, 26}
```

```
{a, b, x, y, z}
```

Transliterate converts to (approximately) equivalent English letters :
(* katakana = alfabeto fonetico, usato per traslitterare le parole straniere *)

```
TableForm[
 Transpose[
  {Alphabet["Greek"],
   Transliterate[Alphabet["Greek"]],
   Transliterate[Alphabet["Greek"], "Japanese"]}
 ]
]
```

α	a	ア
β	b	ブ
γ	g	グ
δ	d	デ
ε	e	エ
ζ	z	ズ
η	e	エー
θ	th	テー
ι	i	イ
κ	k	ク
λ	l	ル
μ	m	ム
ν	n	ン
ξ	x	クス
ο	o	オ
π	p	プ
ρ	r	ル
σ	s	ス
τ	t	テ
υ	y	イ
φ	ph	プー
χ	ch	チ
ψ	ps	プス
ω	o	オー

Rasterize

If you want to, you can also turn text into images, which you can then manipulate using image processing.
Rasterize makes a raster, or bitmap, that is an image gets described in pixels.

Generate an image of a piece of text.

```
imgRaster = Rasterize[ Style["ABC", 50] ]
```

Do image processing on it:

```
EdgeDetect[imgRaster]
```

Vocabulary

“string”	a string
StringLength[“string”]	length of a string
StringReverse[“string”]	reverse a string
StringTake[“string”,4]	take first 4 characters of a string
StringJoin[“string”,”string”]	join strings together
StringJoin[{“string”,”string”}]	join a list of strings
ToUpperCase[“string”]	convert characters to uppercase
Characters[“string”]	convert a string to a list of characters
TextWords[“string”]	list of words from a string
TextSentences[“string”]	list of sentences
WikipediaData[“topic”]	Wikipedia article about a topic
WordCloud[“text”]	word cloud based on word frequencies
WordList[ ]	list of common words in English
Alphabet[]	list of letters of the alphabet
LetterNumber[“c”]	where a letter appears in the alphabet
FromLetterNumber[n]	letter appearing at position n in the alphabet
Transliterate[“text”]	transliterate text into English
Transliterate[“text”,”alphabet”]	transliterate text into other alphabets
RomanNumeral[n]	convert a number to its Roman numeral string
IntegerName[n]	convert a number to its English name string
InputForm[“string”]	show a string with quotes
Rasterize[“string”]	make a bitmap image

Floor

Ceiling, IntegerPart, FractionalPart, Round, Mod, Quotient

TextWords

StringSplit

Exercises

11.1 Join two copies of the string “Hello”.

11.2 Make a single string of the (English) Alphabet, in UpperCase.

11.3 Generate a String of the (English) Alphabet in Reverse order.

11.4 Join 10 copies of the String “MatComp”.

11.5 Use StringTake, StringJoin and Alphabet to get “abcdef” .

11.6 Create a Column with increasing numbers of letters from the string “this is about strings” (IntegerPart).

11.7 Make a BarChart of the Lengths of the words in “A long time ago, in a galaxy far, far away” (TextWords or StringSplit).

11.8 Find the StringLength of the Wikipedia article for “computer”.

11.9 Find how many words are in the Wikipedia article for “computer”.

11.10 Find the first Sentence in the Wikipedia article about “strings”.

11.11 Make a String from the first letters of all Sentences in the Wikipedia article about computers.

11.12 Find the Max word-length among (English) words from WordList[ ].

11.13 Count the number of words in WordList[ ] that start with “q”.

11.14 Make a ListLinePlot of the Lengths of the first 100 words from WordList[ ].

11.15 Use Characters and StringJoin to make a WordCloud of all letters in the words from WordList[ ].

11.16* Use StringReverse to make a WordCloud of the last letters in the words from WordList[ ].

11.17 Find the Roman numerals for the year 1959.

11.18 Find the Maximum StringLength of any Roman-numeral year from 1 to 2021.

11.19 Make a WordCloud from the **first** characters of the Roman numerals up to 10.

11.20 Use Length to find the Length of English, Russian, Italian Alphabets.

11.21 Generate the UpperCase Greek Alphabet.

11.22 Make a BarChart of the Letter Numbers in “compute”.

11.23 Use FromLetterNumber to make a String of 10 Random letters (SeedRandom[25]).

11.24 Make a list of 10 Random 3-letter Strings (SeedRandom[3] and pattern variable ___).

11.25 Transliterate “Your name” into Greek.

11.26 Get the Arabic Alphabet and transliterate it into English.

11.27 Make a White-on-Black size-100 letter “A”.

11.28 Use Manipulate to make an interactive selector of characters (of font size-20) from the Alphabet .

11.29 Use Manipulate to make an interactive selector of Black-on-White outlines of Rasterized size-50 characters from the Alphabet, controlled by a drop-down menu.

11.30 Use Manipulate to create a “vision simulator” that Blurs a size-50 letter “A” by an amount from 0 to 20.

+11.1 Generate a String of the Alphabet followed by the Alphabet written in reverse.

+11.2 Make a Column of a string of the Alphabet and its Reverse.

+11.3 Find how many sentences are in the Wikipedia article for “computer”.

+11.4 Join together without spaces, the words in the first sentence in the Wikipedia article for “strings”.

+11.5 Find the Length of the **longest** word in the Wikipedia article about computers.

+11.6 Plot the lengths of Roman numerals for numbers up to 200.

+11.7 Generate a string by joining the Roman numerals up to 10.

+11.8 Make a ListLinePlot of the successive letter numbers for the concatenation of all Roman numerals up to 10.

+11.9 Find the Maximum StringLength of the Name of any Integer up to 10.

+11.10 Make a list of UpperCase size-20 letters of the Alphabet in RandomColor.

+11.11 Make a list of 10 random 3-letter strings with the Italian alphabet.

+11.12 Create a Manipulate to display Edges of letter A (in size-100) , Blurred from 0 to 30.

+11.13 Add together White-on-Black size-100 letters A and B.

Q & A

What is the difference between “x” and x?

“x” is a string ; x is a symbol (like Plus or Max) that can be defined to actually do computations.

How to enter characters that are not on the keyboard?

You can use whatever methods your computer provides, or constructs such as [\alpha]

How do I put the quotes ” inside a string?

Use \”
If you want to put \” literally , use \\ \”
If you want to put \\ \” literally , use \\ \\ \\ \”

```
stringWithQuotes = "Refer to \"Author\" and others."

stringWithSlashAndQuotes = "Refer to \\\" Author \\\" and others."

stringWith2SlashesAndQuotes = "Refer to \\\\\" Author \\\\\" and others."

stringWith3SlashesAndQuotes = "Refer to \\\\\\\" Author \\\\\\\" and others."
Refer to "Author" and others.

Refer to \" Author \" and others.

Refer to \\" Author \\" and others.

Refer to \\\" Author \\\" and others.
```

How are the colors of elements in word clouds determined?

By default, it is random within a certain color palette. You can specify it, if you want to.

How come the word cloud shows “s” as the most common letter?

“s” is the most common first letter for common **words** in English.
“e” is the most common **letter** .

What about letters that are not English? How are they numbered?

LetterNumber[“α”,“Greek”] gives numbering in the Greek alphabet.
All characters are assigned a character code.
You can find it using **ToCharacterCode**.

What alphabets does *Mathematica* know about?

Basically all the ones that are used today.
Note that when a language uses accented characters, it is sometimes tricky to decide what is “in” the alphabet, and what is just derived from it.

Can I translate words instead of just transliterating their letters?

Yes. Use WordTranslation (see § 35).

```
WordTranslation["translate", "Italian"]
```

```
{tradurre}
```

Tech Notes

RandomWord

RandomWord[10] gives 10 random words. How many of them do you know?

```
SeedRandom[3];
RandomWord[10, Language → "Italian"]
```

```
{oltrepassante, abbreviaste, sospettava, abbozzassero,
 ripenseremo, trasecolava, mutevoli, ferventissimo, sappia, tacciavate}
```

StringTake

StringTake[“string”, -2] takes 2 characters from the end of the string.

```
StringTake["stringhe", -2]
```

```
he
```

ToCharacterCode , FromCharacterCode.

Every character (“a”, “α” or “狼”) is represented by a Unicode character code, found with ToCharacterCode.
You can explore “Unicode space” with FromCharacterCode.

```
{ToCharacterCode["a"], FromCharacterCode[97]}
```

```
{{97}, a}
```

WikipediaData

If you get a different result from WikipediaData, that is because Wikipedia has been changed.

WordCloud

WordCloud automatically removes “uninteresting” words in text, like “the”, “and”, etc.

What about letters that are not English?
Alphabet or Language

If you cannot figure out the name of an alphabet or language, use
CTRL+=
(as described in § 16) to give it in natural language form.

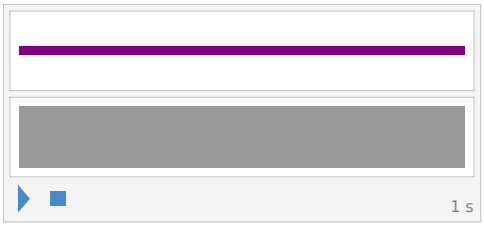
More to Explore

Guide to String Manipulation in *Mathematica*

Sound

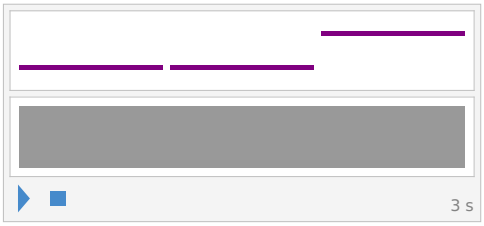
In *Mathematica*, sound works a lot like **Graphics**, but instead of having things like **Circle[ ]** , one has SoundNote.
Press the play button ▷ to actually play sounds.
The default note is "middle C" (i.e. "DO").
The default **duration** (horizontal lenght) of each note is 1 second.
By default, notes sound as if played on a Piano.

```
Sound[ SoundNote[] ]
(* Sound[  SoundNote[ "C" ], 1, "Piano"  ]   *)
```



You can specify a sequence of notes by giving them in a list.

```
(* Sound[  {  SoundNote["C"], SoundNote["C"], SoundNote["G"] }  ] *)
note3 = Map[  SoundNote, {"C" , "C", "G"}  ];
Sound[note3]
```



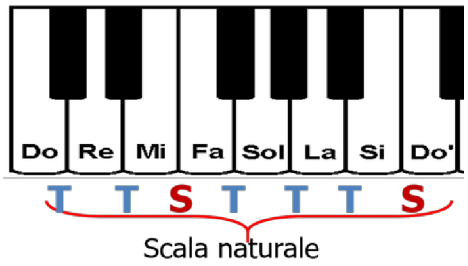
To specify the pitch (height, altezza → nota acuta|grave; frequenza fondamentale dell'onda sonora) of a note, you can also use a number.

```
(* Grandezze fondamentali di una nota:
  altezza, intensità o volume, timbro o purezza *)
http://www3.unisi.it/ricerca/prog/musica/linguaggio/suono.htm
```

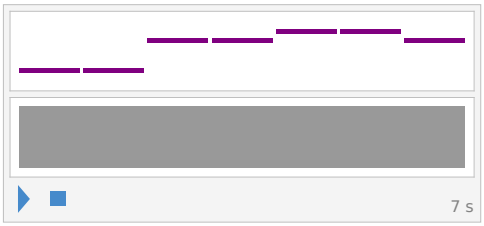
DO (DO centrale, middle C, "C", "C4") is 0 .
Each semitone above the DO goes up by 1 .
SOL (SOL centrale, middle G, "G", "G4") is 7, since SOL is 7 semitones above DO .

```
nd = NotebookDirectory[];
pathOTTAVA = StringJoin[nd, "tono-e-semitono.png"];
imgOTTAVA = Import[pathOTTAVA];
ImageDimensions[imgOTTAVA]
ImageResize[imgOTTAVA, {500, 300}]
```

```
{815, 502}
```



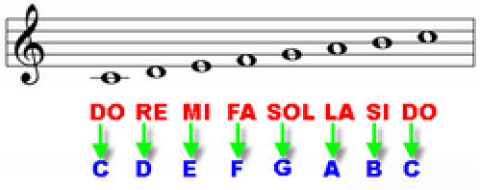

```
note3num = Map[ SoundNote, {0 , 0, 7, 7, 9, 9, 7} ];
(*  DO, DO, SOL, SOL, LA, LA, SOL *)
(* note3num=Map[SoundNote,{"C","C","G","G","A","A","G"}]; *)
Sound[note3num]
```



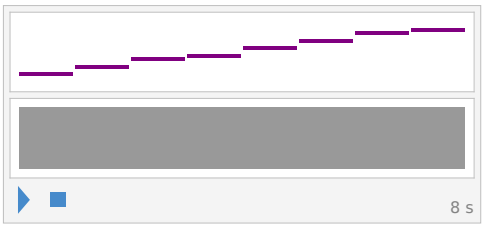
An octave (8 subsequent notes) is 12 semitones.

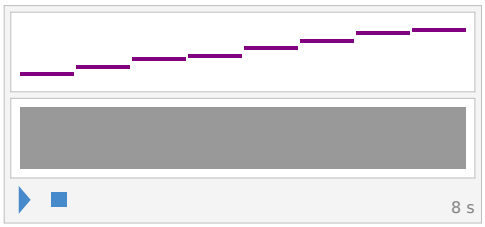
```
nd = NotebookDirectory[];
pathNOTE = StringJoin[nd, "12NOTEnotazioneanglosassone.jpg"];
imgNOTE = Import[pathNOTE];
ImageDimensions[imgNOTE]
ImageResize[imgNOTE, {250, 100}]
```

```
{430, 177}
```



```
(* otto note della scala naturale da DO a DO *)
note8 = Map[SoundNote, {"C" , "D", "E", "F", "G", "A", "B", "C5"}];
note8num = Map[SoundNote, {0 , 2, 4, 5, 7, 9, 11, 12}];
Row[{Sound[note8], "  ", Sound[note8num]}]
```

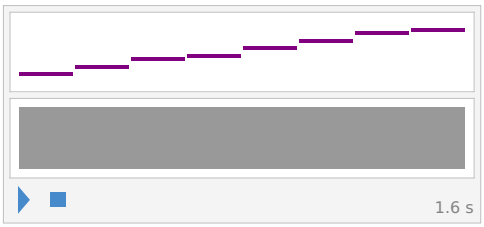




By default, each note lasts 1 second.
Use **SoundNote[pitch, length]** to get a different note-length (durata/*lunghezza della nota*).

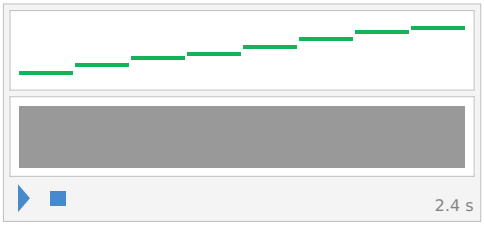
```
noteLen = 0.2 (* ogni nota dura 0.2 second *)
(* DO/"C"/0,  RE/"D"/2,   MI/"E"/4,
   FA/"F"/5,  SOL/"G"/7,  LA/"A"/9,  SI/"B"/11,
   DO/"C5"/12  *)
(* Join  crea {0,2,4,  5,7,9,11,  12} i.e. la scala da DO a DO  *)
notes = Join[
    Table[k, {k, 0, 4, 2}], (* crea {0,2,4} *)
    Table[k, {k, 5, 11, 2}], (* crea {5,7,9,11} *)
    {12}];                  (* singoletto {12} *)
Sound[
 Map[ SoundNote[#, noteLen] &,  notes]
]
```

```
0.2
```



In addition to the Piano, SoundNote can handle various instruments.
The name of each instrument is a string.

```
noteLen = 0.3;
myInstrument = "Guitar";
(* DO/"C"/0,  RE/"D"/2,   MI/"E"/4,
   FA/"F"/5,  SOL/"G"/7,  LA/"A"/9,  SI/"B"/11,
   DO/"C5"/12  *)
notes = Join[
    Table[k, {k, 0, 4, 2}],
    Table[k, {k, 5, 11, 2}],
    {12}];
Sound[
 Map[ SoundNote[#, noteLen, myInstrument] &,  notes]
]
```



Make "random music" (different every time you generate it).
E.G. Play a sequence of 20 notes, with lenght 0.1 and random pitches from 0 to 12:

```
SeedRandom[2];
notes = Table[ RandomInteger[12], 20 ];
noteLen = 0.1;
myInstrument = "Violin";
Sound[
 Map[ SoundNote[#, noteLen, myInstrument] &,  notes]
]
(* SeedRandom[2];
noteLen=0.1;
myInstrument="Violin";
Sound[
 Table[
  SoundNote[  RandomInteger[12],noteLen,myInstrument  ],
  20]
]; *)
```

Vocabulary

Sound[{...}]	create a sound from notes
SoundNote[“C”]	a named note
SoundNote[5]	a note with a numbered pitch
SoundNote[5, 0.1]	a note played for a specified time
SoundNote[5, 0.1, “Guitar”]	a note played on a certain instrument

Map

Play, EmitSound

Button

Outer

MaxIndexed

Exercises

12.1 Generate the sequence of notes with pitches 0, 4, 7 .

12.2 Generate 2 seconds of playing middle A on a **Cello** .

12.3 Generate a “riff” (short musical phrase) of notes, from pitch 0 to 48 in steps of 1, with each note lasting 0.05 seconds .

12.4 Generate a sequence of notes, from pitch 12 down to 0 in steps of 1 (total duration 1.3 sec, i.e. each note lasts 0.1 sec).

12.5 Generate a sequence of 5 notes , starting with DO (0/middle C) and going up by an octave (12 semi-tones) at a time.

12.6 Generate a sequence of 10 notes on a **Trumpet** , with random pitches from 0 to 12 (SeedRandom[8]) and duration 0.2 seconds .

12.7 Generate a sequence of 10 notes, with random pitches from 1 to 12 (SeedRandom[11]), and random durations from 1/10 to 10/10 (tenths of a second).

12.8 Create a Sound from Notes with pitches given by the Integer Digits of 2^31, each playing for 0.1 seconds

12.9 Create a Sound from Notes with pitches given by the Characters in ACCADDE, each playing for 0.3 seconds, sounding like a (violon)Cello.

12.10 Create a Sound from Notes with pitches given by the LetterNumber of characters in your Name or Surname, each playing for 0.1 seconds, sounding like a Harp.

+12.1 Generate a sequence of 3 notes of 1 second of playing middle D, on **Cello, Piano, Guitar** .

+12.2 Generate a sequence of notes, from pitch 0 to 12, going up in steps of 3.

+12.3 Generate a sequence of 5 notes, starting with middle C, then successively going up by a perfect fifth (7 semitones, quinta perfetta/giusta) at a time .

+12.4 Generate 0.02-second notes, with pitches given by the StringLengths of the (Italian) Names of Integers from 1 to 200 (Module).

Q & A

How do I know which instruments are available?

Look at "Details and Options" in the **SoundNote** Help page
(or just start typing and see the completions you are offered).
All the standard MIDI (Musical Instrument Digital Interface) instruments are available.

■ You can also use instrument numbers from 1 to 128.
In *Mathematica*, the first MIDI instrument (Piano) is 1, rather than 0.

```
(* {Sound[SoundNote["A",2,1]],Sound[SoundNote["A",2,"Piano"]]} *)
(* {Sound[SoundNote["A",2,25]],Sound[SoundNote["A",2,"Guitar"]]} *)
{Sound[SoundNote["A", 2, 41]], Sound[SoundNote["A", 2, "Violin"]]}
(* {Sound[SoundNote["A",2,43]],Sound[SoundNote["A",2,"Cello"]]} *)
```

⇒ Styles from 1 to 128, representing MIDI **instruments** (see: ref/format/MIDI)

```
"Accordion"        "Agogo"              "AltoSax"          "Applause"
"Atmosphere"       "Bagpipe"            "Bandoneon"        "Banjo"
"BaritoneSax"      "Bass"               "BassAndLead"      "Bassoon"
"Bird"             "BlownBottle"        "Bowed"            "BrassSection"
"Breath"           "Brightness"         "BrightPiano"      "Calliope"
"Celesta"          "Cello"              "Charang"          "Chiff"
"Choir"            "Clarinet"           "Clavi"            "Contrabass"
"Crystal"          "DrawbarOrgan"       "Dulcimer"         "Echoes"
"ElectricBass"     "ElectricGrandPiano" "ElectricGuitar"   "ElectricPiano"
"ElectricPiano2"   "EnglishHorn"        "Fiddle"           "Fifths"
"Flute"            "FrenchHorn"         "FretlessBass"     "FretNoise"
"Glockenspiel"     "Goblins"            "Guitar"           "GuitarDistorted"
"GuitarHarmonics"  "GuitarMuted"        "GuitarOverdriven" "Gunshot"
"Halo"             "Harmonica"          "Harp"             "Harpsichord"
"Helicopter"       "HonkyTonkPiano"     "JazzGuitar"       "Kalimba"
"Koto"             "Marimba"            "MelodicTom"       "Metallic"
"MusicBox"         "MutedTrumpet"       "NewAge"           "Oboe"
"Ocarina"          "OrchestraHit"       "Organ"            "PanFlute"
"PercussiveOrgan"  "Piano"              "Piccolo"          "PickedBass"
"PizzicatoStrings" "Polysynth"          "Rain"             "Recorder"
"ReedOrgan"        "ReverseCymbal"      "RockOrgan"        "Sawtooth"
"SciFi"            "Seashore"           "Shakuhachi"       "Shamisen"
"Shanai"           "Sitar"              "SlapBass"         "SlapBass2"
"SopranoSax"       "Soundtrack"         "Square"           "Steeldrums"
"SteelGuitar"      "Strings"            "Strings2"         "Sweep"
"SynthBass"        "SynthBass2"         "SynthBrass"       "SynthBrass2"
"SynthDrum"        "SynthStrings"       "SynthStrings2"    "SynthVoice"
"Taiko"            "Telephone"          "TenorSax"         "Timpani"
"Tinklebell"       "TremoloStrings"     "Trombone"         "Trumpet"
"Tuba"             "TubularBells"       "Vibraphone"       "Viola"
"Violin"           "Voice"              "VoiceAahs"        "VoiceOohs"
"Warm"             "Whistle"            "Woodblock"        "Xylophone"
```

■ A percussion event is specified with no pitch

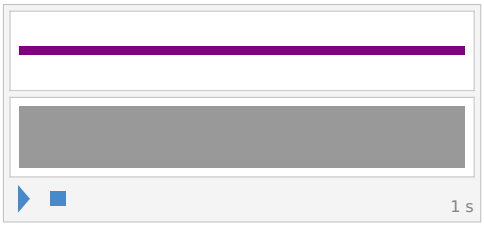
⇒ Possible **percussions** events :

```
"BassDrum"      "BassDrum2"     "BellTree"      "Cabasa"
"Castanets"     "ChineseCymbal" "Clap"          "Claves"
"Cowbell"       "CrashCymbal"   "CrashCymbal2"  "ElectricSnare"
"GuiroLong"     "GuiroShort"    "HighAgogo"     "HighBongo"
"HighCongaMute" "HighCongaOpen" "HighFloorTom"  "HighTimbale"
"HighTom"       "HighWoodblock" "HiHatClosed"   "HiHatOpen"
"HiHatPedal"    "JingleBell"    "LowAgogo"      "LowBongo"
"LowConga"      "LowFloorTom"   "LowTimbale"    "LowTom"
"LowWoodblock"  "Maracas"       "MetronomeBell" "MetronomeClick"
"MidTom"        "MidTom2"       "MuteCuica"     "MuteSurdo"
"MuteTriangle"  "OpenCuica"     "OpenSurdo"     "OpenTriangle"
"RideBell"      "RideCymbal"    "RideCymbal2"   "ScratchPull"
"ScratchPush"   "Shaker"        "SideStick"     "Slap"
"Snare"         "SplashCymbal"  "SquareClick"   "Sticks"
"Tambourine"    "Vibraslap"     "WhistleLong"   "WhistleShort"
```

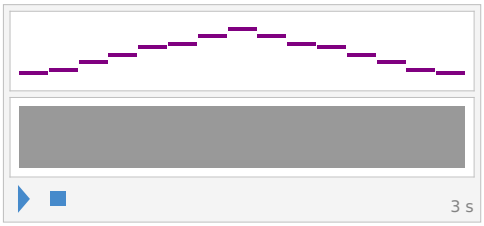
How do I play notes below DO (middle C)?

Use negative numbers.

```
Sound[ SoundNote[-12]]
```



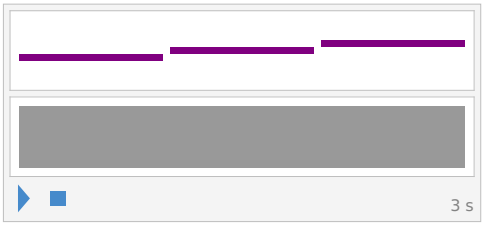

```
(* Form andata = {-12,-11,-9,-7,-5,-4,-2,0}  *)
andata = Union[
    Table[k, {k,  -4,  0, 2}], (* Form {-4,-2,0} *)
    Table[k, {k, -11, -5, 2}], (* Form {-11,-9,-7,-5} *)
    {-12}
  ];
(* Form ar={-12,-11,-9,-7,-5,-4,-2,0,-2,-4,-5,-7,-9,-11,-12} *)
ar = Join[
    Drop[andata, -1],
    Reverse[andata]
  ];

noteLen = 0.2;
Sound[
 Map[  SoundNote[#, noteLen] &,  ar ]
]
```



What are **sharp** (#) and **flat** (b) notes called?

E# (E **sharp/diesis**), Ab (A **flat/bemolle**), etc.
They also have numbers (e.g. E# is 5).
Symbols "#" and "b" can be typed as ordinary keyboard characters
(♯ and ♭ characters are available too).

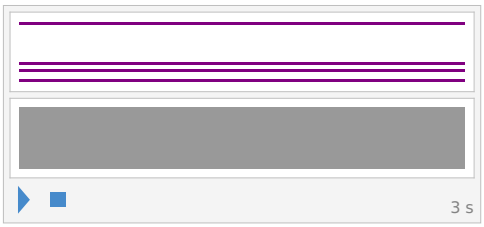
```
(* MI/"E"/4 *)
Sound[{
   SoundNote["Eb"], (* SoundNote[3],*)
   SoundNote["E"] , (* SoundNote[4], *)
   SoundNote["E♯"] (* SoundNote[5] *)
 }]
```



How do I make a chord (accordo)?

Put the names of the Notes in a list

```
(* { RE, FA diesis, LA, RE di due ottave più alto del RE centrale *)
accordo = {"D", "F♯", "A", "D6"};
accordoNum = {2, 6, 9, 26};
noteLen = 3;
Sound[
  SoundNote[accordo, noteLen]
]
(* Equivalentemente: *)
Sound[
   SoundNote[accordoNum, noteLen]
 ];
```



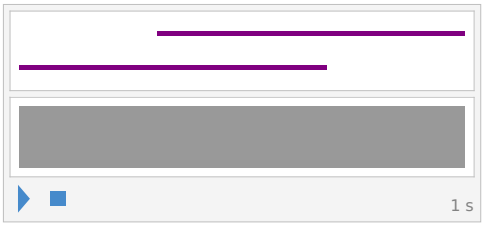
How do I make a rest (or overlap notes)?

■ Use **None** as duration (note Lenght)

```
Sound[{
   SoundNote["C", 0.3],
   SoundNote[None, 0.9],
   SoundNote["G", 0.3]
 }]
```

■ It also possible to overlap notes.

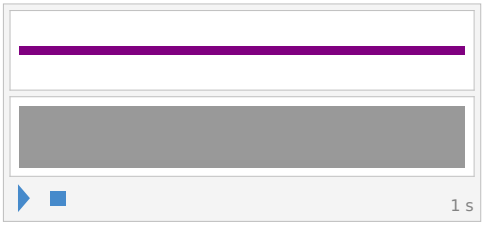
```
Sound[{
   SoundNote["C", {0, 0.7}],
   SoundNote["G", {0.3, 1}]
 }]
```



How do I get a sound to play immediately, without having to press the play button?

■ Use **EmitSound**

```
ssn = Sound[ SoundNote[ ] ]
```




```
EmitSound[ ssn ]
```

```
(* Button creates a button *)
Button[ "play", EmitSound[ ssn ] ]
```

play

- A second example (with Play)
- A third example (Play, EmitSound, Button)
- A more complex example : phone keypad) (Outer[ ] and MapIndexed[ ])

Why do I need quotes in the name of a note like RE (“D”) ?

Because Built-in functions to manipulate Sound expect a string (or a number).
Moreover, typing D (instead of “D”) would be interpreted as a (built-in) function named D[ ].

```
D[y x^2, x]
```

2 x y

Can I record audio and manipulate it?

Yes.
Use **AudioCapture[ ]**
(then use functions like **AudioPlot**, **Spectrogram**, **AudioPitchShift**, etc.)

```
? AudioCapture
```

Symbol

AudioCapture[] creates a temporary interactive interface for capturing an audio signal.
AudioCapture[*file*] captures an audio signal into *file*.

```
$AudioInputDevices
```

{System Setting, Built-in Audio Analog Stereo}

```
(* To start recording, press the Red Button *)
SetDirectory[NotebookDirectory[]];
AudioCapture["myAudio.wav",
 AudioInputDevice → "Built-in Audio Analog Stereo",
 MaxDuration → 10,
 Appearance → "Detailed",
 OverwriteTarget → True]
```

00:00 00:10

Data in File[...dio/myAudio.wav]

Tech Notes

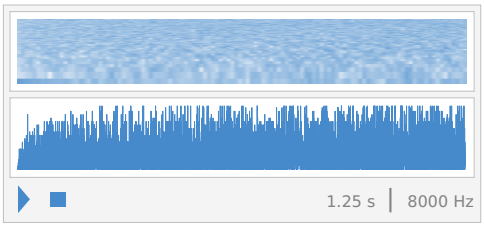
SoundNote is MIDI sound. There are other “sampled sound”, e.g., ListPlay or Audio.

SoundNote corresponds to MIDI (Musical Instrument Digital Interface) sound.
Mathematica also supports “sampled sound”, e.g., using functions like **ListPlay**, or **Audio** (construct that represents all aspects of an audio signal).

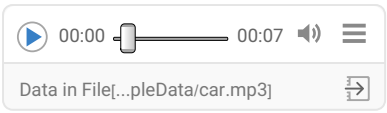
ListPlay[{a1, a2, ... }] creates an object that plays as a sound, whose amplitude is given by the sequence of levels {a1, a2, ...} .

```
(* Play the sequence of differences between Primes 2,3,5,7,11 ....  *)
test = Differences[
    Table[Prime[n], {n, 10 000}]
  ];
Length[test]

ListPlay[ test ]
```

9999




```
(*  egDrums=Import["ExampleData/drums.ogg"];
Audio[egDrums] *)
car = Import["ExampleData/car.mp3"];
Audio[car]
```



To get spoken output, use **Speak**.

```
(* To make a beep, use Beep. *)
(* Does it work ? *)
(* Beep[]  *)

Row[{ Button["press me", Speak["Perfect"]], "  ",
  Button["italiano", Speak["Perfetto"]]}]
```

press me italiano

More to Explore

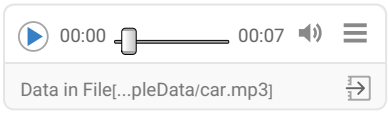
Guide to **Sound Generation** in the *Mathematica*

```
https://reference.wolfram.com/language/guide/SoundAndSonification.html
```

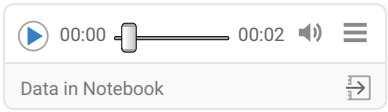
Import ed Export di file audio

```
https://www.wolfram.com/mathematica/new-in-10/enhanced-sound-and-signal-processing/
  import-and-export-mp3-files.html
```

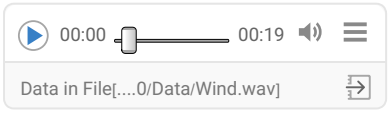
```
(* ref/format/MP3 *)
car = Import["ExampleData/car.mp3"];
Audio[car]
```



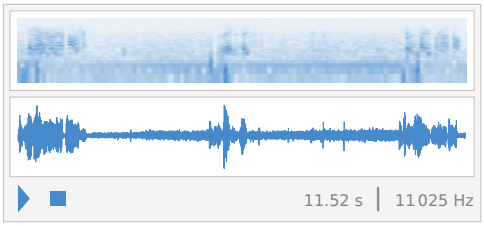

```
(*   FLAC = Free Lossless Audio Codec *)
(* file audio di dimensioni inferiori del 50-60% rispetto ai file audio Wave *)
SetDirectory[NotebookDirectory[]];
Export["bird.flac", ExampleData[{"Audio", "Bird"}]];
bird = Import["bird.flac"];
Audio[bird]
```




```
(* ExampleData["Audio"]; *)
wind = ExampleData[{"Audio", "Wind"}];
Audio[wind]
```



```
(* ExampleData["Sound"]; *)
apollo = ExampleData[{"Sound", "Apollo13Problem"}];
Sound[apollo]
```




```
ExampleData["MachineLearning"];
ExampleData["Statistics"];
```

Programmazione basata su Regole

6.2. Regole e Funzioni *

6.2.3. Definizioni multiple per lo stesso simbolo

Mathematica permette di scrivere definizioni di funzione multiple per uno stesso simbolo, per tenere conto di (possibili ed eventuali) casi differenti; le diverse definizioni diventano regole separate.

Consideriamo qui di seguito una definizione ricorsiva per la funzione fattoriale, in cui abbiamo definito il caso base come regola separata:

```
Clear[fact1];
(* Per definire fact1[0] si puo' usare assegnazione immediata o differita,
dato che, in questo caso, il rhs e' costante *)
fact1[0] = 1;
fact1[n_Integer] := n fact1[n - 1];
DownValues[fact1] // TableForm
(* prima regole  specifiche -- poi  regole  generali  *)
```

```
HoldPattern[fact1[0]] :> 1
HoldPattern[fact1[n_Integer]] :> n fact1[n - 1]
```

```
fact1[3]
```

```
6
```

```
? fact1
```

Symbol

Global`fact1

Definitions

```
fact1[0] = 1

fact1[n_Integer] := n fact1[n - 1]
```

Full Name Global`fact1

Avere piu' definizioni per una stessa funzione puo' sembrare analogo al meccanismo di sovraccarico (overloading) in linguaggi quali C++

[cfr. Bjarne STROUSTRUP. The C++ Programming Language. Addison-Wesley. 4th edition, 2013].
Nella pratica, invece, si rivela essere una risorsa.

Prima di tutto, i pattern di Mathematica possono testare il "tipo" dell'argomento
(cfr. paragrafo 6.3 per una discussione sulla piena generalita' dei pattern).

Meno ovvio, ma forse piu' importante, e' che le differenti alternative non devono essere necessariamente disgiunte.
Notiamo, ad esempio, che zero e' un intero; tuttavia, quando l'argomento della funzione fact1 e' zero, viene applicata la regola (specifica) per fact1[0], non viene applicata la regola (piu' generale) per fact1[n_Integer] .

Questo illustra una strategia generale del motore di match di pattern in Mathematica:
tale motore cerca sempre di applicare le Regole (Rule) piu' specifiche
prima di applicare le Regole piu' generali.

Se non e' ovvio quale regola sia piu' specifica, Mathematica mantiene le regole nell' ORDINE in cui sono state date.

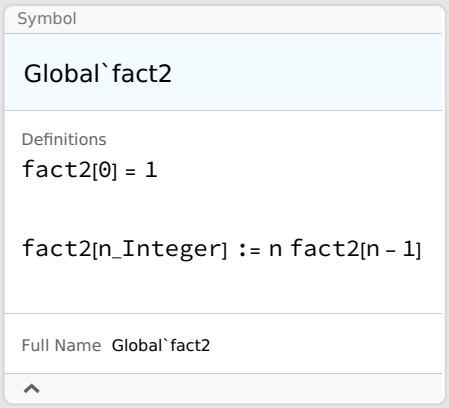
```
Clear[fact2];
fact2[n_Integer] := n fact2[n - 1];
(* anche se inverto l'ordine delle definizioni per fact2,
la regola specifica viene applicata prima di quella generale, come
 mostra la DownValues *)
fact2[0] = 1;
DownValues[fact2] // TableForm
```

```
HoldPattern[fact2[0]] :> 1
HoldPattern[fact2[n_Integer]] :> n fact2[n - 1]
```

```
fact2[3]
```

```
6
```

```
(* anche se inverto l'ordine delle definizioni per fact2,
la regola specifica viene applicata prima di quella generale *)
? fact2
```



DownValues[f] visualizza le Regole per f nell'ordine in cui il Kernel cerca di applicarle.
E' possibile vedere l'ordine in cui le Regole per un simbolo sym sono applicate anche usando la sintassi che segue:

```
? sym
```

In certi casi, *Mathematica* potrebbe non avere sufficienti informazioni per determinare l'ordine corretto di applicazione delle Regole.
A differenza di linguaggi di programmazione logica (e.g. Prolog, cfr. Clocksin W.F. and Mellish C.S., Programming in Prolog. Springer Verlag, Berlin. 5th Edition, 2003), *Mathematica* non implementa il backtracking.
Questo significa che, anche se il Kernel cerca di stabilire quale sia l'ordine migliore di applicazione delle Regole,
se accade che abbia scelto un ordine non corretto, non fa alcun tentativo di seguire un diverso ordine.

Nei casi in cui la scelta dell'ordine di applicazione delle Regole e' ambigua,
possiamo riordinare esplicitamente i DownValues per un determinato simbolo.

Esercizio 1 di 6.2.3

Cancellare (Clear) le definizioni per la funzione fattoriale e ridefinirla, usando la Regola fact[0] dopo la regola fact[n_Integer].
Esaminare l'ordine delle Regole per tale funzione.

```
Clear[fact1];
fact1[0] = 1;
fact1[n_Integer] := n fact1[n - 1];
? fact1
```

```
Global`fact1
```

```
fact1[0] = 1

fact1[n_Integer] := n fact1[n - 1]
```

```
Clear[fact2];
fact2[n_Integer] := n fact2[n - 1];
fact2[0] = 1;
? fact2
```

```
Global`fact2
```

```
fact2[0] = 1

fact2[n_Integer] := n fact2[n - 1]
```

Anche se inverto l'ordine delle definizioni, la Regola specifica fact[0] viene applicata prima della Regola generale fact[n_Integer].

Esercizio 2 di 6.2.3

Programmazione basata su Regole

6.2. Regole e Funzioni *

6.2.5. Rimuovere selettivamente le definizioni

Nello sviluppo di una nuova funzione f puo' accadere che il programmatore commetta un errore ed abbia necessita' di ridefinire la funzione f.
Si puo' usare Clear[f] , che pero' equivale a ripartire dall'inizio.
In alternativa, si puo' rimuovere selettivamente una singola definizione, usando l'operazione di **Unset**, come segue:

```
f[pattern] =.
```

Vediamo un esempio. Consideriamo la funzione fattoriale; su argomenti Interi l'avevamo definita come ricordato qui sotto; questa definizione non va bene per argomenti Reali:

```
fact[0] = 1;
fact[n_Integer] := Apply[Times, Range[n]];
Map[fact, {3.99, 4, 4.01}]
```

```
{fact[3.99], 24, fact[4.01]}
```

Su argomenti Reali, la funzione fattoriale puo' essere definita tramite la funzione speciale di Eulero Gamma[n]; la definizione corretta e':

```
fact[n_] := Gamma[n + 1];
Map[fact, {3.99, 4, 4.01}]
```

```
{23.6415, 24, 24.3645}
```

Definiamo invece erroneamente:

```
fact[n_] := Gamma[n];
fact /@ {3.99, 4., 4.01} (* Map *)
```

```
{5.92519, 6., 6.07593}
```

Controllando la definizione corrente per fact, ci accorgiamo della definizione errata:

```
? fact
```

```
Global`fact
```

```
fact[0] = 1

fact[n_Integer] := Times @@ Range[n]

fact[n_] := Gamma[n]
```

Usiamo la **Unset** per cancellare una sola definizione (quella errata) e controlliamo lo stato corrente delle definizioni associate ad f:

```
fact[n_] =.
? fact
```

```
Global`fact
```

```
fact[0] = 1

fact[n_Integer] := Times @@ Range[n]
```

```
(* Unset       lhs=.    removes any rules defined for lhs
 Scope:  clear values of variables ;  clear functions (downvalues) *)
(*   Clear["context`*"] clears all symbols in a particular context
 Scope:  clear values of variables   ;
clear functions (downvalues and upvalues); clear  several  symbols  *)
(*   ClearAll[symb1, symb2, ...] clears all values,definitions,
attributes,messages,and defaults associated with symbols  *)
```

Nota. In *Mathematica* non possono sussistere definizioni diverse su uno stesso pattern; non avremo quindi modo di definire (ad esempio):

```
fact[0] = 1;
fact[n_Integer] := Apply[Times, Range[n]];


fact[n_] := Gamma[n];
fact[m_] := Gamma[m + 1];
? fact
```

```
Global`fact
```

```
fact[0] = 1

fact[n_Integer] := Times @@ Range[n]

fact[m_] := Gamma[m + 1]
```

Tra le due definizioni, aventi pattern n_ ed m_ **con uguale struttura**, viene mantenuta **l'ultima** ad essere valutata:

```
fact[0] = 1;
fact[n_Integer] := Apply[Times, Range[n]];

fact[m_] := Gamma[m + 1];
fact[n_] := Gamma[n];
? fact
```

```
Global`fact
```

```
fact[0] = 1

fact[n_Integer] := Times @@ Range[n]

fact[n_] := Gamma[n]
```

Un'altra via per modificare una definizione associata ad una funzione f e' tramite DownValues:

```
DownValues[fact] // TableForm
```

```
HoldPattern[fact[0]] :> 1
HoldPattern[fact[n_Integer]] :> Times @@ Range[n]
HoldPattern[fact[n_]] :> Gamma[n]
```

Uso la Drop

```
(* Tolgo dai DonwValues di fact la terza definizione *)
DownValues[fact] = Drop[ DownValues[fact] , {3}]
? fact
```

```
{HoldPattern[fact[0]] :> 1, HoldPattern[fact[n_Integer]] :> Times @@ Range[n]}
```

```
Global`fact
```

```
fact[0] := 1

fact[n_Integer] := Times @@ Range[n]
```

```
? Drop
```

Drop[*list*, *n*] gives *list* with its first *n* elements dropped.

Drop[*list*, –*n*] gives *list* with its last *n* elements dropped.

Drop[*list*, {*n*}] gives *list* with its n^{th} element dropped.

Drop[*list*, {*m*, *n*}] gives *list* with elements *m* through *n* dropped.

Drop[*list*, {*m*, *n*, *s*}] gives *list* with elements *m* through *n* in steps of *s* dropped.

Drop[*list*, seq_1, seq_2, ...] gives a nested list

in which elements specified by seq_i have been dropped at level *i* in *list*. ≫

Programmazione basata su Regole

6.2. Regole e Funzioni *

6.2.6. Programmazione "puramente" basata su regole

Fattoriale

```
Clear[f, n];
f[5] //. {f[0] :> 1, f[n_] :> n f[n - 1]}
(* ReplaceRepeated //. *)
(* f[-1]//. {f[0] :> 1, f[n_] :> n f[n-1] } ; *)
(* Message[ReplaceRepeated: "Exiting after f[-1] scanned 65536 times." ] *)
(* 2^16 == 65536  recursion limit di  ReplaceRepeated *)
```

```
120
```

ReplaceRepeated continua ad applicare le regole (dall'insieme di regole dato) fino a che l'espressione non varia piu'.

NOTA: se adottiamo la formulazione qui sopra, dobbiamo specificare le regole nell'ordine corretto.
Ad esempio, se nel fattoriale invertissimo le due regole, la regola per f[0] non verrebbe mai usata!

```
Clear[f, n];
f[1] //. {f[n_] :> n f[n - 1], f[0] :> 1} // Trace // TableForm
f[5] //. {f[n_] :> n f[n - 1], f[0] :> 1} // Trace // TableForm
(* f[-1]//. {f[n_] :> n f[n-1],f[0] :> 1} ;  *)
(* Message[ReplaceRepeated::"rrlim",f[-1],65536]  *)(* 2^16 == 65536 *)
```

```
f[n_] :> n f[n - 1]                    f[0] :> 1      {f[n_] :> n f[n - 1], f[0] :> 1}
f[n_] :> n f[n - 1]                    f[0] :> 1
f[1] //. {f[n_] :> n f[n - 1], f[0] :> 1}
0
```

```
f[n_] :> n f[n - 1]                    f[0] :> 1      {f[n_] :> n f[n - 1], f[0] :> 1}
f[n_] :> n f[n - 1]                    f[0] :> 1
f[5] //. {f[n_] :> n f[n - 1], f[0] :> 1}
0
```

Notiamo che ad “f” nulla e’ stato assegnato.

```
? f
```

Symbol

Global`f

Full Name Global`f

⌘ In contrasto con la ricorsione, la tecnica di programmazione basata su regole NON da' luogo alla creazione di una grande pila (stack) di valutazioni.
Di conseguenza, tale tipo di programmazione non e' vincolata dalla variabile di sistema $RecursionLimit (che rappresenta una "rete di sicurezza").
Questo fatto puo' rappresentare un vantaggio oppure uno svantaggio, dipendentemente dalle circostanze.

```
$RecursionLimit
```

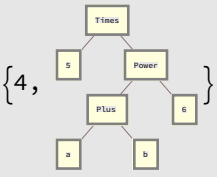
```
1024
```

⌘ Esempio: Depth (e Trace) per vedere quanto e' profonda la pila di valutazioni.

Vediamo un esempio (di programmazione puramente basata su regole) ed usiamo Depth (e Trace) per vedere quanto e' profonda la pila di valutazioni che viene creata).

Prima ricordo che cosa fa Depth:

```
expr = 5 (a + b)^6;
(* Depth fornisce il massimo numero di indici necessario a
 specificare qualsiasi parte di un'espressione, aumentato di 1 *)
{Depth[expr], TreeForm[expr, ImageSize → Tiny]}
? Depth
```



Symbol

Depth[*expr*] gives the maximum
number of indices needed to specify any part of *expr*, plus 1.

Creo una tabella di Fattoriali.

```
(*  Tabella  dei  primi  10  Fattoriali *)
Table[
 f[i] //. {f[0] :> 1, f[n_] :> n f[n - 1]},
 {i, 10}]
```

```
{1, 2, 6, 24, 120, 720, 5040, 40 320, 362 880, 3 628 800}
```

```
(* DIGRESSIONE su Timing *)
(*  Tabella  dei  primi  10  Fattoriali +  Tempo  *)
Timing[
 Table[ f[i] //. {f[0] :> 1, f[n_] :> n f[n - 1]}, {i, 10}]
]
(*  Tempo  per  creare  la  Tabella  dei  primi  10  Fattoriali  *)
Timing[
 Table[ f[i] //. {f[0] :> 1, f[n_] :> n f[n - 1]}, {i, 10}];
]
```

```
{0.000209, {1, 2, 6, 24, 120, 720, 5040, 40 320, 362 880, 3 628 800}}
```

```
{0.000199, Null}
```

Uso Depth (e Trace) per vedere che la pila di valutazioni (relativa alla tabella di Fattoriali) e' profonda 9 :

```
(*  Profondita' di ricorsione per la  Tabella  dei  Fattoriali  *)
Table[
 Trace[f[i] //. {f[0] :> 1, f[n_] :> n f[n - 1]}] // Depth,
 {i, 10}]
```

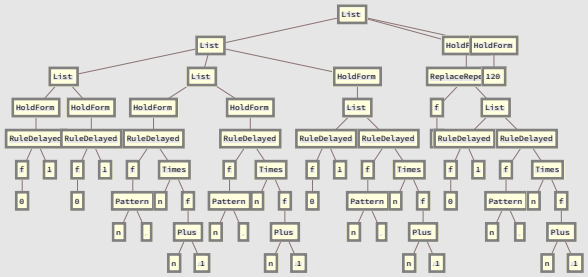
```
{9, 9, 9, 9, 9, 9, 9, 9, 9, 9}
```

```
Timing[Table[
  Trace[f[i] //. {f[0] :> 1, f[n_] :> n f[n - 1]}] // Depth,
  {i, 10}]]
```

```
{0.000363, {9, 9, 9, 9, 9, 9, 9, 9, 9, 9}}
```

Uso TreeForm (e Trace) per visualizzare i 9 livelli della pila di valutazioni (relativa alla tabella di Fattoriali) :

```
Trace[f[1] //. {f[0] :> 1, f[n_] :> n f[n - 1]}] // TreeForm;
Trace[f[5] //. {f[0] :> 1, f[n_] :> n f[n - 1]}] // TreeForm
Trace[f[10] //. {f[0] :> 1, f[n_] :> n f[n - 1]}] // TreeForm;
Trace[f[20] //. {f[0] :> 1, f[n_] :> n f[n - 1]}] // TreeForm;
```



ReplaceRepeated e le altre funzioni di sostituzione ("famiglia Replace") scannerizzano sequenzialmente una lista di regole, alla ricerca di match di pattern.

⌘ Dispatch

Quando la lista di regole e' molto lunga, una operazione di sostituzione ("tipo Replace") puo' essere accellerata usando la funzione **Dispatch**.

```
? Dispatch
```

Symbol

Dispatch[{$lhs_1 \to rhs_1$, $lhs_2 \to rhs_2$, ...}] generates an optimized dispatch table representation of a list of rules. The object produced by Dispatch can be used to give the rules in *expr* /. *rules*.

Dispatch genera una Tabella di Dispaccio (Rapporto, Invio, Spedizione) relativa alla lista di regole.

NOTE.
1. L'uso di Dispatch non ha mai conseguenze sui risultati che vengono ottenuti, mentre puo' velocizzare l'applicazione di una lunga lista di regole.
2. Una lista di regole viene usualmente scannerizzata sequenzialmente, quando viene valutata una espressione del tipo expr /. rules
In alcuni casi, non c'e' bisogno di scannerizzare esplicitamente tutte le regole, ad esempio, quando non e' possibile per tali regole l' essere applicate simultaneamente (e.g. a[1] ⟶ a1 ed a[2] ⟶ a2).
In questo caso, Dispatch genera una Tabella, che usa codici HASH (una funzione di Hash e' non invertibile e mappa una stringa di lunghezza arbitraria in una stringa di lunghezza predefinita; esistono numerosi algoritmi che realizzano funzioni hash con particolari proprietà che dipendono dall'applicazione, e.g. crittografia oppure creazioni di una hash-table per operazioni di ricerca in basi di dati) per specificare

quale sottoinsieme di regole debba effettivamente essere scannerizzato, per una data espressione di input.

3. Una lista di regole generata da assegnazioni realizzate con Set (=) oppure SetDelayed (:=) sono automaticamente ottimizzate con una Tabella di Dispaccio (qualora cio' sia appropriato).

```
rules = {a → b, b → c, c → a, d → e, e → d};
(* Genero una Tabella "dispatch" di Dispaccio per la lista "rules" di Regole *)
dispatch = Dispatch[rules]

(* Il risultato dopo la sostituzione e' lo stesso,
sia che si usi la lista "rules" sia che si usi la tabella "dispatch" *)
{a, b, c, d, e} /. rules
{a, b, c, d, e} /. dispatch
```

```
Dispatch[ Length: 5 ]
```

```
{b, c, a, e, d}
```

```
{b, c, a, e, d}
```

Se la lista di Regole e' molto lunga, usare la Tabella di Dispaccio puo' velocizzare il tempo di esecuzione

```
(* creo una tabella di 10^4 regole *)
rules = Table[
   x[i] → RandomInteger[{1, i}] ,
   {i, 10^4}];
rules // Short

dispatch = Dispatch[rules];
dispatch // Short

Timing[Table[x[i] /. rules, {i, 1000}];]
Timing[Table[x[i] /. dispatch, {i, 1000}];]
```

```
{x[1] → 1, x[2] → 2, x[3] → 3, ≪9995≫, x[9999] → 9549, x[10 000] → 9858}
```

```
Dispatch[ Length: 10 000 ]
```

Data not saved. Save now

```
{0.135565, Null}
```

```
{0.001226, Null}
```

Applichiamo la Dispatch alla funzione fattoriale, programmata con le Regole: in questo caso, non c'e' differenza nei tempi di esecuzione .

```
factRules = {f[0] :> 1, f[n_] :> n f[n - 1]};
factRules // Short

factDispatch = Dispatch[factRules];
factDispatch // Short

Timing[
 Table[
   f[i] //. factRules,
   {i, 10^3}];
]
Timing[
 Table[
   f[i] //. factDispatch,
   {i, 10^3}];
]
```

```
{f[0] :> 1, f[n_] :> n f[n - 1]}
```

```
Dispatch[ <<Length: 2>> ]
```

```
{1.30051, Null}
```

```
{1.47501, Null}
```

(* Per velocizzare i tempi di esecuzione, vedere anche SparseArray e Compile *)

Programmazione basata su Regole

6.3. Mattoni per costruire Pattern *

Mathematica fornisce un ricco insieme di "mattoni" per la costruzione di pattern.
In questo testo non e' possibile illustrarli tutti: verranno presentati solo alcuni esempi.
Per una piu' ampia documentazione, si rimanda a (capitoli successivi di questo testo,
come pure a) molti altri testi della letteratura collegata all'ambiente di Mathematica.

6.3.1 Pattern di vincolo

Tre costrutti possono essere usati come **pattern di vincolo**.
Due di essi vincolano i valori del **Blank singolo** .
Il terzo puo' essere usato per specificare la **relazione** tra differenti variabili—pattern.

_head
Un Blank puo' essere vincolato a combaciare solo con certe espressioni aventi un Head particolare (cfr. § 4.1.3 "Fare un controllo di tipo") appendendo un Head al Blank stesso.

```
(* Il pattern _Integer combacia solo con interi *)
Cases[{1, Sqrt[2], b}, _Integer]
```

```
{1}
```

_?test
Un Blank puo' essere vincolato dalla forma _?test, in cui "test" puo' essere qualsiasi funzione di un solo argomento.
Se l'applicazione del test (alla espressione combaciante col Blank) restituisce il valore True, allora il pattern combacia.

```
(* Secondo modo di scrivere un pattern che combacia solo con interi *)
Cases[{1, Sqrt[2], b}, _?IntegerQ]
```

```
{1}
```

Potendo scegliere, e' piu' efficiente far combaciare la Head strutturalmente usando _Integer piuttosto che usare _?IntegerQ ,
per lo meno in un semplice esempio come quello appena visto.
D'altra parte, esistono molti predicati (cfr. § 3.4) che permettono di testare condizioni complesse,
quali, ad esempio, il fatto che un'espressione sia o meno un Atomo (AtomQ) oppure un numero (NumberQ),
per i quali e' utile la seconda sintassi.

Oltre ad usare predicati definiti da sistema, ovviamente, e' possibile definirne di propri.
Vediamo un esempio.
Costruiamo un predicato "between", che useremo nella forma _?between per testare se un'espressione e' un numero tra 1 e 10.

```
(* Costruisco una funzione da usare nella Cases in forma di predicato *)
between[x_] := 1 ≤ x ≤ 10
(* costruisco una tabella di 20 numeri random tra 1 e 100 *)
SeedRandom[8];
tabella = Table[ RandomReal[{1, 100}], {20} ];
Sort[tabella]
(* Qui il pattern _?between combacia con un'
 espressione solo se essa e' un numero tra 1 e 10.
 Quindi Cases cerca, nella tabella, eventuali numeri tra 1 e 10 *)
Cases[ tabella , _?between]
```

```
{1.88952, 6.8186, 14.7483, 16.476, 18.684, 18.6946,
 20.7307, 21.4937, 22.1198, 29.5493, 29.5974, 45.4095, 48.6137,
 49.0859, 49.9518, 75.7534, 80.5866, 89.1365, 89.7225, 91.3095}
```

```
{1.88952, 6.8186}
```

E' possibile costruire un pattern che **combini assieme** il far combaciare la Head (e.g. _Integer) con una funzione di test (e.g. _? NonNegative).
Nell'esempio che segue, il pattern combacia solo con interi non negativi.

```
test = {-3, -1, 0, 1, 1.5 , 2, 2.5};
Cases[ test , _Integer?NonNegative]
```

```
{0, 1, 2}
```

Nell'esempio che segue, il pattern combacia solo con reali non negativi.

```
SeedRandom[8];
tabella2 = Table[ RandomReal[{-50, 50}], {20}] ;
Sort[tabella2]
Cases[ tabella2 , _Real?NonNegative]
```

```
{-49.1015, -44.1226, -36.1128, -34.3677, -32.1374, -32.1267,
 -30.07, -29.2993, -28.6669, -21.1623, -21.1137, -5.14196, -1.90533,
 -1.42839, -0.553752, 25.5085, 30.3905, 39.0268, 39.6187, 41.2217}
```

```
{39.6187, 39.0268, 30.3905, 41.2217, 25.5085}
```

pattern /; condition

Una condizione consiste in una chiamata alla Condition (/;) seguita da una qualsiasi espressione che coinvolga variabili—pattern.

Essa puo' essere attaccata, praticamente, a qualsiasi parte di un pattern.

Ad esempio, una qualsiasi delle regole che seguono puo' essere usata per definire (in modo ricorsivo) la funzione fattoriale (per numeri interi e non negativi):

```
body[n_] := Apply[ Times , Range[n]];

factorial1[n_Integer?NonNegative] := body[n]

factorial2[n_Integer /; NonNegative[n]] := body[n]

factorial3[n_Integer] /; NonNegative[n] := body[n]

factorial4[n_Integer] := body[n] /; NonNegative[n]

factorial5[n_] := body[n] /; IntegerQ[n] && NonNegative[n]

{factorial1[n], factorial2[n], factorial3[n], factorial4[n], factorial5[n]} /. n → 5
```

```
{120, 120, 120, 120, 120}
```

Le Condition sono utili perche' permettono di ovviare alla necessita' di scrivere una funzione test separata.

Ad esempio, la funzionalita' della "between" (che testa una lista di numeri per considerare solo quelli compresi tra 1 e 10)

puo' essere incorporata (come Condition) nella definizione di una funzione, come segue.

```
(* Cosi' definita, la funzione f accetta solo argomenti tra 1 e 10 ,
quindi resituisce il fattoriale solo per tali argomenti *)
Clear[body, f, x, n, tabella];

body[n_] := Apply[ Times , Range[n]];

f[ x_ /; 1 ≤ x ≤ 10] := body[x];

SeedRandom[2];
tabella = Module[{ntemp, ftemp},
    Table[
     ntemp = RandomInteger[{-10, 10}];
     ftemp = f[ntemp];
     {ntemp, ftemp},
     {5}]
  ];
TableForm[tabella, TableAlignments → Center]
```

```
-7     f[-7]
 8    40320
 1       1
 0      f[0]
 7     5040
```

```
(* Nota.  Ricordo  che  il  fattoriale  di  0  vale  1  *)
Factorial[0]
```

```
1
```

Le Condition sono piu' flessibili dei costrutti *?test* ,
perche' esse possono coinvolgere piu' di una sola variabile—pattern alla volta,
come mostrato nell'esempio che segue.

```
Clear[body, f, x, y, n];
body[x_] := Apply[ Times , Range[x]];
(* La funzione f  verra'  chiamata solo se il suo
 secondo argomento e' maggiore del primo argomento  *)
f[x_, y_] /; x < y := body[x];

SeedRandom[2];
tabella = Module[{rtemp, stemp, ftemp},
   Table[
    rtemp = RandomInteger[{1, 10}];
    stemp = RandomInteger[{1, 20}];
    ftemp = f[rtemp, stemp];
    {rtemp, stemp, ftemp},
    {5}]
  ];
TableForm[tabella, TableAlignments → Center]
```

```
9    19    362 880
6     4     f[6, 4]
1    14       1
2     2     f[2, 2]
4    10       24
```

Nota.

Nel caso particolare dell'esempio qui sopra, non e' possibile piazzare Condition cosi':

```
f[x_, y_ /; x < y ] := body[x]; (* NO! *)
```

Cosi' facendo, il Kernel assume che noi si voglia testare la variabile—pattern y_ paragonandola ad un simbolo globale x
(notiamo i colori verde e blu dei vari simboli),
anziche' paragonare tra loro gli argomenti x ed y .

```
Clear[body, f, x, y, n];
body[x_] := Apply[ Times , Range[x]];
(* La funzione f verra'  chiamata solo se
 il suo secondo argomento e' maggiore non del primo,
bensi' di una variabile x globale, che pero' non e' settata *)
f[x_, y_ /; x < y] := body[x];
SeedRandom[2];
tabella = Module[{rtemp, stemp, ftemp},
   Table[
    rtemp = RandomInteger[{1, 10}];
    stemp = RandomInteger[{1, 20}];
    ftemp = f[rtemp, stemp];
    {rtemp, stemp, ftemp},
    {5}]
  ];
tabella // TableForm
```

```
9    19    f[9, 19]
6    4     f[6, 4]
1    14    f[1, 14]
2    2     f[2, 2]
4    10    f[4, 10]
```

- Unico **vincolo su una condizione**: se la condizione coinvolge piu' di una sola variabile—pattern alla volta,
allora la condizione stessa deve apparire fuori dalla piu' piccola espressione contenente tutte tali variabili—pattern.

```
(* SI! *) f[x_, y_] /; x < y  := body[x];
(* NO! *) f[x_, y_ /; x < y ] := body[x];
```

Per ragioni sia di efficienza, sia di leggibilita', e' meglio piazzare una condizione il piu' vicino possibile alla/alle variabile/i che la condizione stessa coinvolge.

Tests and Conditionals

Test whether 2 + 2 is equal to 4 :

```
2 + 2 == 4
```

```
True
```

Test whether 2×2 is greater than 5 :

```
2 + 2 > 5
```

```
False
```

If

The conditional function **If**[*test* , *x* , *y* , *z*] gives:
“x” then-branch (*test* is True) ,
“y” else-branch (*test* is False),
“z” otherwise-branch (*test* is neither True nor False, i.e., Indeterminate).

```
(* In statement 1, test is True ⟹ result is x *)
If[ 2 + 2 == 4, x, y, z]
```

```
x
```

```
(* Unequal  ≠ ,typed as " != " *)
(* In statement 2, test is False ⟹ result is y *)
If[2 + 2 ≠ 4, x, y, z]
```

```
y
```

```
(* Since a,b are not Set, in statement 3,
test is neither True nor False ⟹ result is z *)
If[a < b, x, y, z]
```

```
z
```

If and TrueQ

```
(* TrueQ forces the condition to return a Boolean value *)
TrueQ[a < b]
(* Even  with  a,b not  Set, test TrueQ[a<b] is  False ⟹ result is y *)
If[TrueQ[a < b], x, y, z]
```

```
False
```

```
y
```

If and Map

By using **Map** (/@), we can apply **If** (as a pure function) to every element of a list.

```
(* Map a test on a list *)
mylist = {1, 2, 3, 4, 5, 6, 7, 8, 9};
Map[ If[# < 4, x, y] & , mylist ]
(* equivalently *)
If[# < 4, x, y] & /@ mylist;

(* another test on mylist *)
Map[ If[# ≠ 4, x, y] & , mylist]
```

{x, x, x, y, y, y, y, y, y}

{x, x, x, y, x, x, x, x, x}

Select

It is often useful to Select elements in a list that satisfy a test.
You can do this with **Select**, giving your test as a pure function.

```
(* Select from a list those element that verify a test *)
mylist = {1, 2, 3, 4, 5, 6, 7, 8, 9};
Select[ mylist , # > 3 & ]
Select[ mylist , 2 ≤ # ≤ 5 & ]
```

{4, 5, 6, 7, 8, 9}

{2, 3, 4, 5}

Select and EvenQ, IntegerQ, PrimeQ, ...

Beyond size comparisons like <, > , ==, *Mathematica* includes many other kinds of tests:
EvenQ , **OddQ** , **IntegerQ, PrimeQ ....**
("Q" indicates that the functions are asking a question.)

```
{EvenQ[4], EvenQ[5], OddQ[4], OddQ[5]}
```

{True, False, False, True}

```
(* Select Even numbers/Primes from a list *)
mylist = {1, 2, 3, 4, 5, 6, 7, 8, 9};
Select[mylist, EvenQ]
Select[mylist, PrimeQ]
```

{2, 4, 6, 8}

```
{2, 3, 5, 7}
```

```
(*  Select Integers from a list  *)
mylist2 = {1., 2., 3., 4., 5, 6., 7, 8, 9};
Select[mylist2, IntegerQ]
(* Forsee  the  output of the following  Select *)
Select[mylist2, EvenQ];
Select[mylist2, PrimeQ];
```

```
{5, 7, 8, 9}
```

Select with AND, OR, NOT

To combine tests:
&& represents And
|| represents Or
! represents Not

```
mylist = {1, 2, 3, 4, 5, 6, 7, 8, 9};
(*  Select Even elements  greater than 2 *)
Select[mylist, EvenQ[#] && # > 2 &]
```

```
{4, 6, 8}
```

```
(* Select elements that are NOT either "Even" OR ">4"
  i.e. Select Odd elements ≤ 4 *)
s1 = Select[mylist, !(EvenQ[#] || # > 4) &]
s2 = Select[mylist, OddQ[#] && # ≤ 4 &];
s1 == s2
```

```
{1, 3}
```

```
True
```

Select , LetterQ, LetterNumber

There are many other "Q functions" that ask various kinds of questions.
LetterQ tests whether a string consists entirely of letters.

```
(* The space between letters is not a letter; nor is "!" *)
test = {"ab", "a  b", "!", "2", "$", "#", "@", "%", "&", "+"};
Map[ LetterQ, test ]
```

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

```
(* Turn a string into a list of Characters,then test which are letters *)
char = Characters["30 is the best!"]
Map[LetterQ , char]
(* Select the Characters that are letters *)
Select[char, LetterQ]
```

```
{3, 0,  , i, s,  , t, h, e,  , b, e, s, t, !}
```

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

```
{i, s, t, h, e, b, e, s, t}
```

```
(* Task:  Select letters that appear after Position 13, strictly,
in the UK and Italian Alphabets : "o" is at Position 15 and 13, respectively *)
char = Characters["our test"];
Select[char, LetterQ[#] && LetterNumber[#] > 13 &  ]
Select[char, LetterQ[#] && LetterNumber[#, "Italian"] > 13 &]
```

```
{o, u, r, t, s, t}
```

```
{u, r, t, s, t}
```

Select and WordList

You can use Select to find words, in (English and) Italian, that are palindromes.

```
Select[ WordList[],                    StringReverse[#] == # &];
Select[WordList[Language → "Italian"], StringReverse[#] == # &]
```

```
{aerea, afa, ala, alla, ama, ara, atta, avallava, aveva, ebbe, elle,
 ere, esse, ingegni, inni, ivi, non, onorarono, oro, oso, osso, otto}
```

Select and MemberQ

MemberQ tests whether something appears as an element, or member of a list.

```
MemberQ[{1, 3, 5, 7}, 5]
```

```
True
```

```
(* Select pairs containing  x  *)
Select[
  {{1, y}, {2, x}, {3, z}, {x, 5}, {y, 6}, {z, 7}},
  MemberQ[#, x] &
]
(* Select entries containing  x  *)
Select[
  {{1, y}, {2, x}, {3, z}, {x, 5}, {z, 6},
   {1, y, 4}, {2, x, y}, {z, 2, y},
   {4, 1, x, t}, {x, 1, x, t}},
  MemberQ[#, x] &
]
```

```
{{2, x}, {x, 5}}
```

```
{{2, x}, {x, 5}, {2, x, y}, {4, 1, x, t}, {x, 1, x, t}}
```

SelectFirst

```
(*  Select numbers in the Range 50 to 100, whose digit sequences contain 2 *)
Select[
 Range[50, 100],
 MemberQ[ IntegerDigits[#], 2] &
]
```

```
{52, 62, 72, 82, 92}
```

```
(* Equivalently, in this particular case *)
Select[
  Range[50, 100],
  Last[ IntegerDigits[#]] == 2 &
 ];

(*  Select the First of the numbers in the Range 50 to 100,
whose digit sequences contain 2  *)
SelectFirst[
 Range[50, 100],
 MemberQ[IntegerDigits[#], 2] &
]
```

```
52
```

Cases

```
(* Equivalently,  with Cases,
form all even numbers in [50,100], then Take the second one *)
even = Cases[
   Range[50, 100],
   _?EvenQ
 ]
(*  The construction   pattern?test  applies a function test to the whole
 expression matched by pattern, to determine whether there is a match  *)
Take[even, {2}]
First[%]
```

```
{50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70,
 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100}
```

```
{52}
```

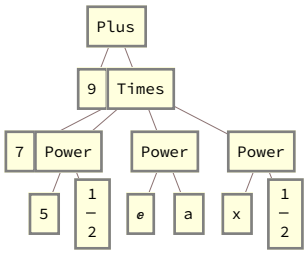
```
52
```

Cases$\left[\left\{e_1, e_2, \ldots\right\},\ \textbf{pattern}\right]$ gives a list of those e_i that match the **pattern**

The first argument to **Cases** does not need to have List as Head (default level is 1)

```
expr = 7 Sqrt[5 x] Exp[a] + 9
TreeForm[expr]
```

$9 + 7\ \sqrt{5}\ e^{a}\ \sqrt{x}$



```
(* default level is 1 *) Cases[ expr , _Integer ]
(* all levels *)Cases[ expr , _Integer, Infinity ]
(* level 3 only *) Cases[ expr, _Integer, {3} ]
(* levels from 1 to 2 included *) Cases[expr , _Integer, 2 ]
```

{9}

{9, 7, 5}

{5}

{9, 7}

ImageIstanceQ and Entity

■ **ImageIstanceQ** is a function based on machine-learning,
that tests whether an Image is an Instance of a particular kind of something
(e.g., a cat).

ImageInstanceQ[**image** , **category**] tests whether **image** is an instance of **category**

```
(* Test if an image is that of a cat *)
```

```
ImageInstanceQ[ <image> ,  Entity["Species", "Species:FelisCatus"]]
```

True

■ **Entity** is a symbolic expression, built-in, in the sense that *Mathematica* knows about thousands of types of real-world entities:
Cities, Countries, Chemicals, Movies, Populations, Satellites, Species, Airports Companies , etc.

```
Entity["City", List["Rome", "Lazio", "Italy"]]
```

Rome

■ To enter an **Entity** in input , use CTRL = (Shift += , under Cloud), that yields a box in which it is possible to type in English (e.g. capital of Italy) then evaluate the cell

Italy COUNTRY [capital city]

Rome

```
FullForm[%]
```

```
Entity["City", List["Rome", "Lazio", "Italy"]]
```

- Examples with Species (cats).

```
Entity["Species", "Species:FelisCatus"]
```

domestic cat

Geographic examples with Select, Entity, **Position**, GeoDistance

Here is a geographic example of Select :
find which cities, in a list, are **less** than 3000 miles from San Francisco.

```
(* Do  not put a space in compound names  *)
(*  city, area/region, state *)
testCity = Entity["City", {"SanFrancisco", "California", "UnitedStates"}];
worldCities = {
    Entity["City", {"London", "GreaterLondon", "UnitedKingdom"}],
    Entity["City", {"Madrid", "Madrid", "Spain"}],
    Entity["City", {"Tokyo", "Tokyo", "Japan"}],
    Entity["City", {"Chicago", "Illinois", "UnitedStates"}]
  };

Select[
 worldCities,
 GeoDistance[#, testCity] < Quantity[3000, "Miles"] &
]
```

{ Chicago }

```
(* Do  not put a space in compound names  *)
italianCities20list = {
    {"Sulmona", "Abruzzes"},
    {"Monopoli", "Apulia"},
    {"Rotonda", "Basilicata"},
    {"Tropea", "Calabria"},
    {"Caserta", "Campania"},
    {"Bologna", "EmiliaRomagna"},
    {"Pordenone", "FriuliVeneziaGiulia"},
    {"Rome", "Lazio"},
    {"Alassio", "Liguria"},
    {"Sondrio", "Lombardy"},
    {"Urbino", "Marche"},
    {"Termoli", "Molise"},
    {"Ivrea", "Piemonte"},
    {"Olbia", "Sardegna"},
    {"PiazzaArmerina", "Sicily"},
    {"Volterra", "Toscana"},
    {"Pinzolo", "TrentinoAltoAdige"},
    {"Gubbio", "Umbria"},
    {"Aosta", "ValleDAosta"},
    {"Garda", "Veneto"}
   };

italianCities20 = Map[
   Entity["City", Flatten[List[#, "Italy"]]] &,
   italianCities20list]
```

{ Sulmona , Monopoli , Rotonda , Tropea , Caserta , Bologna ,
Pordenone , Rome , Alassio , Sondrio , Urbino , Termoli , Ivrea ,
Olbia , Piazza Armerina , Volterra , Pinzolo , Gubbio , Aosta , Garda }

⟶ **Position**[*expression*, *pattern*] gives a list of the Positions at which objects matching *pattern* appear in *expression*.

```
testRome = Entity["City", {"Rome", "Lazio", "Italy"}];
positionRome = Flatten[
   Position[ italianCities20, testRome ]
 ]
```

{8}

```
(* Drop  Rome  from  the  list  of  Italian Cities  *)
italianCities19 = Drop[italianCities20, positionRome];
```

```
(* Find which Italian cities, in our current list, are <200km from Rome *)
testRome = Entity["City", {"Rome", "Lazio", "Italy"}];

Select[
 italianCities19,
 GeoDistance[#, testRome] < Quantity[200, "Kilometer"] &
]
```

{ Sulmona , Caserta , Gubbio }

Vocabulary

a==b test for equality
a<b tests whether less
a>b tests whether greater
a≤b tests whether less or equal
a≥b tests whether greater or equal
If[test, u, v] gives u if test is True, and v if False
Select[list,f] selects elements that pass a test
EvenQ[x], OddQ[x] tests whether even /odd
IntegerQ[x] tests whether an integer
PrimeQ[x] tests whether a prime number
LetterQ[string] tests whether there are only letters
MemberQ[list,x] tests whether x is a member of list
ImageInstanceQ[image,category] tests whether image is an instance of category
Entity
Cases
Position
/; Condition
ContainsNone
ArrayPlot
StringTake, StringStartsQ, StringEndsQ
&& And , || Or , LogicalExpand
Mod

Exercises

28.1 Test whether 123^321 is greater than 456^123.

28.2 Get a list of numbers up to 100 whose IntegerDigits Total up to be < 5.

28.3 Make a list of the first 20 integers, where Prime numbers are Styled Red.

28.4 Find words in WordList[ ] that begin and end with the letter "p".

28.4bis Find words in WordList[ ], in Italian, that begin and end with the letter "n".

28.5 Make a list of the first 100 Primes, keeping only those whose last IntegerDigit is < 3 (Array).

28.6 Find RomanNumerals up to 50 that do **not** contain "I".

28.7 Get a list of RomanNumerals up to 50 that are palindromes (and have more than 1 letter).

28.8 Find Names of Integers up to 100 that begin and end with the same letter.

28.8bis Find Italian Names of Integers up to 100 that begin and end with the same letter.

28.9 Get a list of Words, longer than 18
Characters, from the Wikipedia article on "words".

28.10 Starting from 1000, divide by 2 if the number is Even, compute 3 # + 1 & if the number is Odd; do this 111 times (Collatz problem 1937 : this process converges to 1, regardless of which positive integer is chosen initially, with an appropriate number of iterations. Named after the German mathematician Lothar Collatz, 1910-1990).

28.11 Make a WordCloud of 5-letter words in the Wikipedia article on computers.

28.12 Find words in WordList[ ], whose first 3 letters are the same as their last 3 read backward, and that are not palindrome.

28.13 Find all 12-letter words in WordList[ ] for which the Total of LetterNumber values is 100.

+28.1 Make a Table of Integers up to 25, where every integer ending in 3 is replaced with 0.

+28.2 Use Table and If to make a 5×5 array that is 1 on its leading diagonal, and 0 otherwise.

+28.3 Get a list of numbers up to 1000 that are equal to 1 Mod 7 and also equal to 1 Mod 8.

+28.4* Make a list of numbers up to 50, where multiples of 3 are replaced by Black ■, multiples of 5 by White □ , and multiples of 3 and 5 by Red ■ (swatch).

+28.5 Use Select to get a list of planets whose mass is larger than Earth.

+28.6 Make a 50×50 **ArrayPlot**, in which a pixel (square) at position (i, j) is Black if Mod[i, j]==0, otherwise is White.

+28.7 Make a 100×100 ArrayPlot, in which a pixel (square) is Black if the values of both its x and y positions do not contain a 5.

Q & A

When do I need to use parentheses with &&, ||, etc.?

There is an order of operations that is an analog of Arithmetic.
And[ ] && is like Times
Or[ ] || is like Plus
Not[ ] ! is like Subtract or Minus, i.e., -x is Times[-1, x]

Furthermore, the Laws of De Morgan (British mathematician, 1806-1871) are valid:
Not (A And B) is (Not A) Or (Not B)
Not (A Or B) is (Not A) And (Not B).

If in doubt, use **LogicalExpand**

```
(* This means (NOT p)AND q *)
!p && q ;
LogicalExpand[%]
```

```
q && ! p
```

```
(* This means NOT(p AND q)
 which is (NOT P)OR(NOT q) *)
!(p && q);
LogicalExpand[%]
```

```
! p || ! q
```

What are some other “Q” query functions?

Are there other ways to find real-world entities with certain properties than using Select?

Tech Notes

More to Explore

Parts of Lists

Part lets you pick out an element of a **List**.

```
mylist = {a, b, c, d, e, f, g};
(* Pick out element 2 *)
Part[mylist, 2];
(* [[...]] is an alternative notation *)
mylist[[2]]
```

b

Negative part numbers count from the end of a list

```
mylist = {a, b, c, d, e, f, g};
(* Pick out element 2 from the end *)
Part[mylist, -2]
```

f

You can ask for a list of parts, by giving a list of part numbers.

```
mylist = {a, b, c, d, e, f, g};
(* Pick out parts 2, 4,5 *)
Part[mylist, {2, 4, 5}]
```

{b, d, e}

;; lets you ask for a span or sequence of parts.

```
mylist = {a, b, c, d, e, f, g};
(* Pick out parts 2 through 5 *)
Part[mylist, 2 ;; 5]
```

{b, c, d, e}

Take also allows to get elements of a list

```
mylist = {a, b, c, d, e, f, g};
(*  Take the first 4 elements from a list  *)
Take[mylist, 4]
(*  Take the last 2 elements from a list  *)Take[mylist, -2]
```

{a, b, c, d}

{f, g}

Drop also allows to discard elements of a list

```
mylist = {a, b, c, d, e, f, g};
(*  Discard  the first 4 elements from a list  *)
Drop[mylist, 4]
(*  Discard the last 2 elements from a list  *)Drop[mylist, -2]
```

```
{e, f, g}
```

```
{a, b, c, d, e}
```

List of lists , i.e., arrays and matrices

Each sublist acts as a row in the array.

```
myarray = {{a, b, c}, {d, e, f}, {g, h, i}};
Grid[myarray]
(*  MatrixForm  *)
(*  TableForm  *)
```

```
a b c
d e f
g h i
```

```
myarray = {
    {a, b, c},
    {d, e, f},
    {g, h, i}
  };
TreeForm[myarray];
(* Pick out row 2 *)
Part[myarray, 2]
(* Pick out element 1 on row 2 *)
Part[myarray, 2, 1]
```

```
{d, e, f}
```

```
d
```

Row permutation with Part .

```
myarray = {{a, b, c}, {d, e, f}, {g, h, i}};
(* Permute rows 3 and 1 *)
permuteRows31 = Part[myarray, {3, 2, 1}];
Map[MatrixForm, {myarray, permuteRows31}]
```

$$\left\{\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}, \begin{pmatrix} g & h & i \\ d & e & f \\ a & b & c \end{pmatrix}\right\}$$

It is also often useful to pick out **columns** in an array.

```
myarray = {{a, b, c}, {d, e, f}, {g, h, i}};
(* Get column 2 *)
Part[myarray, All, 2]
```

```
{b, e, h}
```

Column permutation with Part .

```
myarray = {{a, b, c}, {d, e, f}, {g, h, i}};
(* Permute columns 3 and 1 *)
permuteCols31 = Part[myarray, All, {3, 2, 1}];
Map[MatrixForm, {myarray, permuteCols31}]

(* permuteCols31transposed=Transpose[
  Part[  Transpose[myarray]  ,{3,2,1}]
 ];
permuteCols31==permuteCols31transposed*)
```

$$\left\{\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}, \begin{pmatrix} c & b & a \\ f & e & d \\ i & h & g \end{pmatrix}\right\}$$

Position

The function Position finds the list of positions at which something appears.

```
myarray = {{a, b, c}, {d, e, f}, {g, h, i}};
(* There is only one d,
and it appears at position 2,1 *)
Position[myarray, d]
```

```
{{2, 1}}
```

```
myxyz = { {x, y, x}, {y, y, x}, {x, y, y}, {x, x, y} };
(* This gives a list of all positions at which x appears *)
Position[myxyz, x]
```

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

Positions at which "m" (small letter) occurs in a list of characters :

```
char = Characters["Computational Mathematics"];
Flatten[ Position[char, "m"]]
```

```
{3, 20}
```

Find the positions at which 0 occurs in the digit sequence of 2^43

```
id = IntegerDigits[2^43]
Flatten[ Position[id, 0]]
```

```
{8, 7, 9, 6, 0, 9, 3, 0, 2, 2, 2, 0, 8}
```

```
{5, 8, 12}
```

ReplacePart

The function ReplacePart lets you replace parts of a list

```
mylist = {a, b, c, d, e, f, g};
(* Replace  Part 3  of  mylist  with  x *)
ReplacePart[mylist, 3 → x]
(* Replace  two  components of  mylist *)
ReplacePart[mylist, {3 → x, 5 → y}]
```

```
{a, b, x, d, e, f, g}
```

```
{a, b, x, d, y, f, g}
```

Replace 5 randomly chosen parts with "--"

```
SeedRandom[2];
char = Characters["Computational Mathematics"];
len = Length[char];
ReplacePart[
 char,
 Table[ RandomInteger[ len ] → "--",  5 ]
 (* Substitute the n-th Character of char ,
 where  n is given by a RandomInteger in [1,len] *)
]
```

```
{C, o, --, p, u, t, a, t, i, --, --, a, l,  , M, a, t, --, e, m, a, t, --, c, s}
```

Nothing

You may want particular parts of a list to just disappear.
Do this by replacing them with **Nothing**.

```
{1, 2, Nothing, 4, 5, Nothing, 7}
```

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

```
mylist = {a, b, c, d, e, f, g};
(* Replace  two  components  with Nothing *)
ReplacePart[mylist, {1 → Nothing, 3 → Nothing}]
```

```
{b, d, e, f, g}
```

Consider 50 Random Words, dropping those longer than 5 characters, and reversing others :

```
SeedRandom[11];
rand50words = RandomSample[ WordList[Language → "Italian"] , 50 ];
Map[
 If[ StringLength[#] > 5,  Nothing,  StringReverse[#]  ] &,
 rand50words
]
```

{nocid, iaug}

TakeLargest , TakeLargestBy

Take takes a specified number of elements in a list based on their position.
Take**Largest** (and TakeSmallest) takes elements based on their size.
TakeLargest**By** (and TakeSmallestBy) takes elements based on applying a function.

```
(*  Take the 5 largest elements in [1,20] *)
TakeLargest[ Range[20], 5]
```

{20, 19, 18, 17, 16}

```
(* From the first 100 Roman numerals,
 take 5 based on the largest StringLength  *)
roman100 = Array[RomanNumeral, 100];
sol = TakeLargestBy[ roman100 , StringLength , 5 ]
(*  Map[StringLength,sol];  *)
FromRomanNumeral[sol]
```

{LXXXVIII, LXXXIII, XXXVIII, LXXVIII, LXXXVII}

{88, 83, 38, 78, 87}

Vocabulary

Part[list,n] part n of a list
list[[n]] short notation for art n of a list
list[[{n1,n2,...}]] list of parts n1, n2, ...
list[[n1;;n2]] span (sequence) of parts n1 through n2
list[[m,n]] element from row m, column n of an array
list[[All,n]] all elements in column n
Take[list,n] take the first n elements of a list
TakeLargest[list,n] take the largest n elements of a list
TakeSmallest[list,n] take the smallest n elements of a list
TakeLargestBy[list,f,n] take elements largest by applying f
TakeSmallestBy[list,f,n] take elements smallest by applying f

Position[list,x] all positions of x in list
ReplacePart[list,n→x] replace part n of list with x
Nothing a list element that is automatically removed

Exercises

31.1 Find the last 5 digits in 2^1000.

31.2 Pick out letters 10 through 20 in the Italian alphabet.

31.3 Make a list of the letters at even-numbered positions in the Italian alphabet.

31.4 Make a ListPlot (Joined) of the penultimate Digit in the first 100 powers of 12 (i.e. in 12^k with k=1,...., 100).

31.5 Join lists of the first 20 squares and cubes, and Take the 10 Smallest elements of the combined list.

31.6 Find the positions of the word “software” in the Wikipedia entry for “computers”.

31.7 Make a Histogram of the Positions of the letter “e” inside the words of WordList[ ].

31.8* Make a list of the first 30 cubes, in which every cube whose Position is a square is replaced by Red.

31.9* Make a list of the first 20 Primes, using **Nothing** to Drop those whose first Digit is < 5 .

31.10 Make a Grid: the first row is Range[10]; then, create 9 more rows, randomly removing an element in each new row.

31.11 Find the first three longest words in WordList[ ].

31.12 Find the 3 longest IntegerNames, in Italian, for Integers up to 100.

31.13 Find the 3 Italian names for integers up to 100, that have the largest number of “e” in them.

Q & A

What happens if one asks for a part of a list that does not exist?

We get a message, and the original computation is returned unevaluated.

```
mylist = {a, b, c, d, e, f, g};
Part[mylist, 10]
```

Part: Part 10 of {a, b, c, d, e, f, g} does not exist.

```
{a, b, c, d, e, f, g}[[10]]
```

Can I just get the first position at which something appears in a list?

Yes. Use FirstPosition.

```
mylist1 = {a, b, a, a, b, c, b};
FirstPosition[mylist1, b]
```

```
{2}
```

```
mylist2 = {{a, a, b}, {b, a, a}, {a, b, a}};
FirstPosition[mylist2, b]
```

```
{1, 3}
```

Tech Notes

First , Last are equivalent to [[1]] and [[-1]].

```
mylist = {a, b, c, d, e, f, g};
First[mylist] == Part[mylist, 1]
Last[mylist] == Part[mylist, -1]
```

```
True
```

```
True
```

In specifying parts, 1;;-1 is equivalent to All.

```
mylist = {a, b, c, d, e, f, g};
Part[mylist, 1 ;; -1] == Part[mylist, All]
```

```
True
```

More to Explore

Guide to **Parts** of **Lists** in *Mathematica*

```
Button["Guide to Parts and Lists",
 SystemOpen["https://reference.wolfram.com/language/guide/ElementsOfLists.html"]]
```


```
Hyperlink["https://reference.wolfram.com/language/guide/ElementsOfLists.html"]
```

https://reference.wolfram.com/language/guide/ElementsOfLists.html

Patterns

Patterns are a fundamental concept in *Mathematica*.
The pattern _ is called "blank" and stands for “anything”.

MatchQ

MatchQ tests whether something matches a **pattern**.

```
(* pattern { _ , x , _ } is matched
 by any list of 3 elements with x in the middle *)
Clear[a, b, c, d, f, x, l1, l2, l3, mq1, mq2, mq3, pattern];
pattern = {_ , x , _};
l1 = {a, x, b};                    mq1 = MatchQ[ l1 , pattern ];
l2 = { {a, c} , x , {b, d, f} };   mq2 = MatchQ[ l2 , pattern ];
l3 = { {a, b, c}, x, b };          mq3 = MatchQ[ l3 , pattern ];
{mq1, mq2, mq3}
```

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

```
(* No match below, since the 2nd element is not x *)
Clear[a, b, c, z, x, l4, l5, l6, l7, mq4, mq5, mq6, mq7];
pattern = {_ , x , _};
l4 = {a, z, b};          mq4 = MatchQ[ l4 , pattern ];
l5 = {a, b, x};          mq5 = MatchQ[ l5 , pattern ];
l6 = {{a, x, c}, a, b};  mq6 = MatchQ[ l6 , pattern ];
l7 = {a, {a, x, c}, b};  mq7 = MatchQ[ l7 , pattern ];
{mq4, mq5, mq6, mq7}
```

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

```
(* No match, since there are more than 3 elements *)
Clear[a, b, c, d, x, l8, mq8];
pattern = {_ , x , _};
l8 = {a, b, x, c, d}; mq8 = MatchQ[ l8 , pattern ]
```

```
False
```

```
(* Pattern {_,_} is matched by any list of 2 elements *)
Clear[a, b, c, d, f, l1, l2, l3, mq1, mq2, mq3, newpattern];
newpattern = {_, _};
l1 = {a, a};                mq1 = MatchQ[l1, newpattern];
l2 = { {a, b, c}, {d, f} };  mq2 = MatchQ[l2, newpattern];
(* No  match, since  there are more than 2  elements *)
l3 = {a, a, a};  mq3 = MatchQ[l3, newpattern];
{mq1, mq2, mq3}
```

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

Cases (1)

MatchQ lets you test one thing at a time against a pattern.
Cases lets you pick out all the elements ("cases") in a list that match a pattern.

```
Clear[a, b, c, testList, pattern0, pattern1, pattern2, pattern3];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a}, {b, b, b}, {b}};
```

```
(*  Find all elements that match pattern {_,_}  *)
pattern0 = {_ , _};
Cases[  testList,  pattern0 ]
```

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

```
(* Blank:
 Find all elements that match   { b , one element } *)
pattern1 = {b , _};
Cases[  testList,  pattern1 ]
```

```
{{b, a}, {b, b}}
```

```
(* Double Blank:
  Find all elements that match { b ,  some elements } *)
pattern2 = {b , __};
Cases[  testList,  pattern2 ]
```

```
{{b, a}, {b, b}, {b, b, b}}
```

```
(*  Triple  Blank:
  Find all elements that match   { b ,  optional } *)
pattern3 = {b , ___};
Cases[  testList,  pattern3 ]
```

```
{{b, a}, {b, b}, {b, b, b}, {b}}
```

MatchQ can be used to test each element and see if it matches { b , _ }

```
Clear[a, b, c];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a}, {b, b, b}, {b}};
pattern1 = {b, _};

map1 = Map[
  MatchQ[#, pattern1] &,
  testList]
```

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

Select

Select what **match**es gives the same result as **Cases**

```
(* Here, we use the same testList as before, same pattern1,
and  same pure function used in map1:  MatchQ[ # ,pattern1 ] & *)
Clear[a, b, c, testList, sel];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a}, {b, b, b}, {b}};
pattern1 = {b, _};
(* Select picks out  elements of testList having a True MatchQ with  pattern1 *)
sel = Select[
   testList,
  MatchQ[#, pattern1] &
 ]
(* Cases picks out  elements of testList matching  pattern1 *)
sel == Cases[  testList,  pattern1  ]
```

```
{{b, a}, {b, b}}
```

```
True
```

Cases (2)

In a pattern, a | b indicates "either a **Or** b" .
It is called **Alternatives**[a, b] (see: guide/Patterns) .
It is a pattern object that represents any of the patterns a, b

```
Clear[a, b, c, testList, pattern];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a}, {b, b, b}, {b}};
(* pattern = list of 2 elements, with  a|b  as first element *)
pattern = {a | b, _};
Cases[  testList,  pattern  ]
```

```
{{a, a}, {b, a}, {b, b}}
```

Note (see Q&A below).
Alternatives[a, b] is different from Or[a,b] i.e. a||b

Or[a,b, ...] is used to define/combine assertions, assumptions, conditions, systems of equations/inequalities, sets given by algebraic conditions
(see: guide/LogicAndBooleanAlgebra)

```
(*  Or[a,b] is a logical operator .
  It evaluates its arguments in order, giving:
 True  immediately if any of them are True ;
False  if they are all  False. *)
```

Example

Create a **list**, then pick out elements that match particular **pattern**s.

```
(* List:  3-digit numbers from 100 to 500, with step 55 *)
input = Range[100, 500, 55]
digits = IntegerDigits[ input ]
```

```
{100, 155, 210, 265, 320, 375, 430, 485}
```

```
{{1, 0, 0}, {1, 5, 5}, {2, 1, 0}, {2, 6, 5}, {3, 2, 0}, {3, 7, 5}, {4, 3, 0}, {4, 8, 5}}
```

```
(* Pattern:  3-digit numbers whose last digit is 5 *)
idPattern5 = {_, _, 5};
output5 = Cases[ digits, idPattern5 ]
Map[ FromDigits , output5]
```

```
{{1, 5, 5}, {2, 6, 5}, {3, 7, 5}, {4, 8, 5}}
```

```
{155, 265, 375, 485}
```

```
(* another Pattern:  3-digit numbers whose second digit is 1 or 2 *)
idPattern12 = {_, 1 | 2, _};
output12 = Cases[ digits, idPattern12 ]
Map[ FromDigits , output12]
```

```
{{2, 1, 0}, {3, 2, 0}}
```

```
{210, 320}
```

Single, Double, Triple Blank

```
Clear[a, b, c, d, testList, pattern1, pattern2, pattern3];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a},
    {a, c, b}, {b, b, b}, {a}, {b}, {c}, {d}, {}};
(* find  sequences  ending  with  b  ..... *)
(*  .... and beginning with 1 element *)
pattern1 = {_, b};
Cases[  testList,  pattern1 ]
(*  .... and beginning with 1 or more  elements *)
pattern2 = {__, b};
Cases[  testList,  pattern2 ]
(*  .... and beginning with none, 1 or more  elements *)
pattern3 = {___, b};
Cases[  testList,  pattern3 ]
```

```
{{b, b}}
```

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

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

```
Clear[a, b, c, d, testList, patternC1, patternC2, patternC3];
testList = {{a, a}, {b, a}, {a, b, c}, {b, b}, {c, a},
    {b, b, b}, {a}, {b}, {c}, {d}, {}};
(* find  sequences ending  with a or b ,
                or beginning with  c  ... *)
(*  .... and  with another element *)
patternC1 = {_, a | b} | {c, _};
Cases[  testList,  patternC1 ]
(*  .... and  with  1 or more other elements *)
patternC2 = {__, a | b} | {c, _};
Cases[  testList,  patternC2 ]
(*  .... and  with other optional elements *)
patternC3 = {___, a | b} | {c, __};
Cases[  testList,  patternC3 ]
```

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

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

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

Pattern Head[_]

Patterns are not just about lists; they can involve anything.

```
Clear[a, b, c, f, g, x, y, test, patternF1, patternF2, patternF3];
test = {f[1], f[2, 3], g[1], g[2, 3],
    {a, b, c}, {a, f, c}, {}, {a, f[y], c},
    f[x], f[x, x], f[], f[a],
    g[x], g[x, x], g[], g[a]};
(* Pick out cases that match  pattern f[_] *)
patternF1 = f[_];
Cases[test, patternF1]
(* Pick out cases that match  pattern f[__] *)
patternF2 = f[__];
Cases[test, patternF2]
(* Pick out cases that match  pattern f[___] *)
patternF3 = f[___];
Cases[test, patternF3]
```

```
{f[1], f[x], f[a]}
```

```
{f[1], f[2, 3], f[x], f[x, x], f[a]}
```

```
{f[1], f[2, 3], f[x], f[x, x], f[], f[a]}
```

```
Clear[a, b, c, f, g, h, x, y, test, patternH1, patternH2, patternH3];
test = {f[1], f[2, 3], g[1], g[2, 3],
   {a, b, c}, {a, f, c}, {}, {h}, h, {a, f[y], c},
   f[x], f[x, x], f[], f[a],
   g[x], g[x, x], g[], g[a]};
(* Pick out cases  matching  pattern _[_] *)
patternH1 = _[_];
Cases[test, patternH1]
(* Pick out cases   matching pattern _[__] *)
patternH2 = _[__];
c2 = Cases[test, patternH2]
(* Pick out cases  matching  pattern _[___] *)
patternH3 = _[___];
c3 = Cases[test, patternH3]
(*  patternH3  matches everything  in the  test, except for atom  h *)
test
Sort[test] == Sort[Append[c3, h]]
```

```
{f[1], g[1], {h}, f[x], f[a], g[x], g[a]}
```

```
{f[1], f[2, 3], g[1], g[2, 3], {a, b, c}, {a, f, c}, {h}, {a, f[y], c}, f[x], f[x, x], f[a], g[x], g[x, x], g[a]}
```

```
{f[1], f[2, 3], g[1], g[2, 3], {a, b, c}, {a, f, c},
 {}, {h}, {a, f[y], c}, f[x], f[x, x], f[], f[a], g[x], g[x, x], g[], g[a]}
```

```
{f[1], f[2, 3], g[1], g[2, 3], {a, b, c}, {a, f, c}, {},
 {h}, h, {a, f[y], c}, f[x], f[x, x], f[], f[a], g[x], g[x, x], g[], g[a]}
```

```
True
```

ReplaceAll (/.)

One of the many uses of patterns is to define replacements

```
Clear[a, b, mylist, myrule];
mylist = {a, b, a, a, b, b, a, b} ;
(* myrule =  Rule[b, Red] *)
myrule =  b → Red;
(* ReplaceAll[   mylist ,   myrule  ]   *)
mylist /. myrule
```

{a, ■, a, a, ■, ■, a, ■}

```
Clear[a, b, c, mylist, myrule];
mylist = {{1, a}, {2, a}, {1, b}, {1, a, b}, {1, b, c}, {a, 1}, {b, 1}} ;
myrule = {1, _} → Red;
mylist /. myrule
(* ReplaceAll[   mylist ,   myrule  ]   *)
```

{■, {2, a}, ■, {1, a, b}, {1, b, c}, {a, 1}, {b, 1}}

```
Clear[a, b, c, mylist, myrules];
mylist = {{1, a}, {1, b}, {1, c}, {1, a, b}, {2, b, c}, {2, b}, {1}, {b}, {c}} ;
myrules = { {1, _} → Red,
            {__, b} → Yellow ,
           {___, c} → Orange } ;
mylist /. myrules
(* ReplaceAll[   mylist ,   myrules  ]   *)
```

{■, ■, ■, ■, ■, ■, {1}, {b}, ■}

```
Clear[a, b, c, mylist, myrules, myrules2];
mylist = {{1, a}, {1, b}, {1, c}, {1, a, b}, {2, b, c}, {2, b}, {1}, {b}, {c}} ;
myrules = { {1, _} → Red,
            {__, b} → Yellow ,
           {___, c} → Orange } ;
myrules2 = RotateRight[myrules] ;
TableForm[myrules2]
(* TableForm[myrules] mostra in che ordine vengono eseguite le sostituzioni *)
ReplaceAll[ mylist ,  myrules2 ]
(* myrules3=RotateLeft[myrules] ; *)
(* ReplaceAll[   mylist ,   myrules3 ]; *)
```

{__, c} →
{1, _} →
{__, b} →

{, , , , , , {1}, {b}, }

Named_pattern

The "blank" pattern _ matches anything.
For example, {_, _} matches any list of two elements.
But what if you want to insist that the two elements must be the same?
You can do that using a pattern like {x_, x_}.

```
Clear[a, b, c, mytest, mypatternAny, mypattern, mypatternX];
mytest = {{a, a, a}, {a, a}, {a, b}, {a, c},
   {b, a}, {b, b}, {a}, {b}, {c}, {x, x}, {{a}, {b}}, {{a}, {a}}, {{x}, {x}}};
mypatternAny = {_, _};
mypattern = {x_, x_};
mypatternX = {x, x};
Cases[mytest, mypatternAny]
Cases[mytest, mypattern]
Cases[mytest, mypatternX]
```

{{a, a}, {a, b}, {a, c}, {b, a}, {b, b}, {x, x}, {{a}, {b}}, {{a}, {a}}, {{x}, {x}}}

{{a, a}, {b, b}, {x, x}, {{a}, {a}}, {{x}, {x}}}

{{x, x}}

x_ is an example of a **named** pattern.
Named patterns are important in replacements,
because they give a way for making use of Parts of what one is replacing.

```
Clear[a, mycolors, mypatternYellow];
a = 3;
(* swatches *)
mycolors = { {1, Red},
             {1, Blue},
             {1, Cyan, Pink},
             {Purple, 1, Brown},
             {Green, Magenta, 1},
             {2, Orange} ,
             {1, a},
             {1, 4},
             {1, {1, 0}},
             {1, b},
             {1, b} /. b → 5
   (* , HoldForm[{1,b}/.b→5] *)}
(* Replace 2-component sublists of the form {1,x} with {x,x,Yellow,x,x} *)
mypatternYellow = {1, x_} → {x, x, Yellow, x, x};
output = ReplaceAll[ mycolors, mypatternYellow ]
```

{{1, ■}, {1, ■}, {1, ■, ■}, {■, 1, ■}, {■, ■, 1}, {2, ■}, {1, 3}, {1, 4}, {1, {1, 0}}, {1, b}, {1, 5}}

{{■, ■, ■, ■, ■}, {■, ■, ■, ■, ■}, {1, ■, ■}, {■, 1, ■}, {■, ■, 1}, {2, ■}, {3, 3, ■, 3, 3}, {4, 4, ■, 4, 4}, {{1, 0}, {1, 0}, ■, {1, 0}, {1, 0}}, {b, b, ■, b, b}, {5, 5, ■, 5, 5}}

Rule

The form **x → b** is **Rule[x, b]** (* RuleDelayed :→ ossia :> *)
If x_ appears on the left-hand side of a Rule ,
then whatever x_ matches can be referred to (on the right - hand side) as x

```
Clear[a, f, g];
functions = { f[1] , g[2] , f[2] , g[3], f[], f[2, 3] } ;
myrule = f[x_] → x + 10 ;
(* functions /. myrule *)
ReplaceAll[ functions , myrule ]
(* f[1] → 1 + 10 *)
(* g[2] → no match with f[ x_ ] *)
(* f[2] → 2 + 10 *)
(* g[3] → no match with f[ x_ ] *)
(* f[ ] → no match with f[ x_ ] *)(* f[2,3] → no match with f[ x_ ] *)
```

{11, g[2], 12, g[3], f[], f[2, 3]}

You can use Rules inside **Cases** as well.
Below, Cases picks out elements in the list that match f[x_] ,

and gives the result of replacing them by x+10:

```
Clear[f, g];
functions = { f[1] , g[2] , f[2] , g[3], f[], f[2, 3] };
myrule = f[x_] → x + 10 ;
Cases[ functions , myrule ]
```

```
{11, 12}
```

Vocabulary

_	pattern standing for anything (“blank”)
__	pattern standing for any sequence (“double blank”)
___	pattern standing for any optional sequence (“triple blank”)
x_	pattern named x
a|b	pattern matching a or b
MatchQ[expr,pattern]	test whether expr matches a pattern
Cases[list,pattern]	find cases of a pattern in a list
lhs→rhs	**Rule** for transforming lhs into rhs
expr/.lhs→rhs	**ReplaceAll** lhs by rhs in expr
HoldForm	
MaxIterations	
Nothing	
RotateRight	

Exercises

Q & A

Tech Notes

If /. (ReplaceAll) is followed by a digit , you must leave a space before the digit, to avoid confusion with division

```
{
 ReplaceAll[x + 2 a,  2 a → 1],
 x + 2 a /.  2 a → 1,
 x + 2 a /. {2 a → 1}
}
```

```
{1 + x, 1 + x, 1 + x}
```

```
(* With no space, it becomes a division  x+2a/(.2)a → 1  *)
(* It  is  interpreted  as
                x + 2a/(0.2)    a → 1
      that is   x + 2a(1/(0.2)) a → 1
      that is   x + 2a(5.)      a → 1   *)
```

```
x + 2 a / .2 a → 1
```

```
10. a^2 + x → 1
```

More to Explore

Guide to Patterns in *Mathematica*.

```
Hyperlink["https://reference.wolfram.com/language/guide/Patterns.html"]
```

https://reference.wolfram.com/language/guide/Patterns.html

Immediate and Delayed Values

There are two ways to assign a value to something in *Mathematica* :

- **Set** , i.e., immediate assignment (=)
- **SetDelayed** , i.e., delayed assignment (:=)

```
(* to define equations Equal == *)
```

In immediate assignment, the value is computed immediately when the assignment is done, and is never recomputed.

In delayed assignment, the computation of the value is delayed, and is done every time the value is requested.

For example, consider the difference between

valueSet = RandomColor[]

and

valueDelayed:=RandomColor[]

With **Set** , a random color is immediately generated :

```
SeedRandom[3];
valueSet = RandomColor[]
```

Every time you ask for valueSet, you get the same (random) color :

```
valueSet
```

```
valueSet
```

With **SetDelayed** , no random color is immediately generated:

```
SeedRandom[3];
valueDelayed := RandomColor[]
```

Every time you ask for valueDelayed, **RandomColor[]** is evaluated, and a new color is generated:

```
valueDelayed
```

```
valueDelayed
```

```
{valueSet, valueDelayed}
```

{■, ■}

It is very common to use **SetDelayed** if something is not ready yet when you are defining a value.

For example, you can make a **SetDelayed** for **Circle** even though it does not yet have a value :

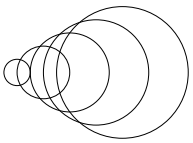
```
(* Here, n is not set *)
circles := Graphics[
          Table[ Circle[{x, 0}, x / 2], {x, n} ],
   ImageSize → Tiny ]
```

```
circles
```

Table: Iterator {x, n} does not have appropriate bounds.

Once **n** is given a value , you can ask for circles:

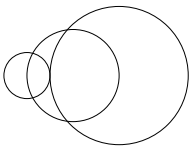
```
n = 5; circles
```



We can define a function of **n**

```
circlesN[n_] := Graphics[
   Table[ Circle[{x, 0}, x / 2], {x, n} ],
   ImageSize → Tiny ]
```

```
circlesN[3]
```



```
? circlesN
```

Symbol

Global`circlesN

Definitions

```
circlesN[n_] := Graphics[Table[Circle[{x, 0}, x/2], {x, n}], ImageSize → Tiny]
```

Full Name Global`circlesN

```
(*  fib[0]=1;
    fib[1]=1;
    fib[n_]:=fib[n]=fib[n-1]+fib[n-2];
fib[10]  *)
```

In delayed assignment, we do not compute a value until we need it.

There is a notion of immediate and delayed Rules too :
- **Rule** , i.e., immediate rule (→ , typed ->)
- **RuleDelayed** , i.e., delayed rule (:→ , typed :>)

In an immediate rule **x → rhs** , rhs is evaluated immediately.

In a delayed rule, **x :→ rhs** , rhs is re-evaluated every time it is requested.

For example, consider the following immediate rule, where a specific value for RandomReal[ ] is immediately computed:

```
(* SeedRandom[3]; *)
x → RandomReal[]
Rule[x, RandomReal[]  ]
```

```
x → 0.182328
```

```
x → 0.648071
```

You can **Replace** four **x**' s, but they will all be the same :

```
SeedRandom[3];
{x, x, x, x}  /.  x → RandomReal[]
ReplaceAll[ {x, x, x, x} , Rule[x, RandomReal[]  ] ]
```

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

```
{0.00869692, 0.00869692, 0.00869692, 0.00869692}
```

This is a Delayed Rule, where the computation of RandomReal[] is delayed:

```
(*  SeedRandom[3];  *)
x :→ RandomReal[]
RuleDelayed[x, RandomReal[]  ]
```

```
x :→ RandomReal[]
```

```
x :→ RandomReal[]
```

RandomReal[] is computed separately when each x is Replaced, giving four different values :

```
SeedRandom[3];
{x, x, x, x} /. x :→ RandomReal[]
ReplaceAll[ {x, x, x, x} ,  RuleDelayed[x, RandomReal[]  ] ]
```

```
{0.478554, 0.00869692, 0.347029, 0.13928}
```

```
{0.180603, 0.528701, 0.578587, 0.760349}
```

Vocabulary

x := value	**SetDelayed**	delayed assignment, evaluated every time x is requested
x :→ value	**RuleDelayed**	delayed rule, evaluated every time x is encountered (typed :>)

Exercises

39.1 Replace **x** in {x, x+1, x+2, x^2} by the same Random Integer up to 100.

```
SeedRandom[3];
{x, x + 1, x + 2, x ^ 2} /. x → RandomInteger[100]
```

```
{61, 62, 63, 3721}
```

39.2 Replace each **x** in {x, x+1, x+2, x^2} by a separately chosen Random Integer up to 100.

```
SeedRandom[3];
{x, x + 1, x + 2, x ^ 2} /. x :→ RandomInteger[100]
```

```
{61, 33, 82, 81}
```

Q & A

Why not always use SetDelayed ?

Because you do not want to recompute things unless it is necessary.

It is more efficient to just compute something once, then use the result over and over again.

What happens if I do x=x+1, with x not having a value?

Do not do it !
You would start an infinite loop, that will eventually get cut off by the system.
Note that x = {x} is the same story.

Significance of inputs and outputs labeled
In[n]:=
Out[n]=

It indicates that inputs are assigned to **In[n]** and outputs to **Out[n]**.
The **:=** for input means the assignment is delayed, so that if you ask for **In[n]** , the result will be recomputed.

More about Patterns

_ ("blank") stands for anything
x_ ("x blank") stands for anything, but calls it x
_h stands for anything with head h
x_h stands for anything with head **h**, and calls it **x**

▹ Define a function whose argument is an Integer, named **n** :

```
digitback[n_Integer] := Framed[ Reverse[IntegerDigits[n]] ]
```

Function **digitback[n]** evaluates whenever the argument is an Integer

Notes. IntegerDigits[0] yields {0}
For n∈N, IntegerDigits[-n] yields {n} i.e. IntegerDigits discards the Sign of n

```
testlist = {1234, -6712, x, {4, 3, 2}, y /. y → 3, 2^32, 22., 0};
(*  digitback[n] only evaluates if n is Integer *)
Flatten[{
  Map[ digitback, testlist],
  digitback[2, 2],
  digitback[]
 }]
```

{{4, 3, 2, 1}, {2, 1, 7, 6}, digitback[x], digitback[{4, 3, 2}], {3},
{6, 9, 2, 7, 6, 9, 4, 9, 2, 4}, digitback[22.], {0}, digitback[2, 2], digitback[]}

▹ Sometimes you may want to put a **Condition** (**/;**) on a pattern.
For example, **n_Integer /; n > 0** means "any integer that is greater than 0"

Define a function whose argument is an Integer named **n** , with n > 0 :

```
pdigitback[n_Integer /; n > 0] := Framed[Reverse[IntegerDigits[n]]]
```

```
testlist = {1234, -6712, x, {4, 3, 2}, y /. y → 3, 2^32, 22., 0};
(*  digitback[n] only evaluates if n>0 is Integer *)
Flatten[{
  Map[pdigitback, testlist],
  pdigitback[2, 2],
  pdigitback[]
 }]
```

```
{{4, 3, 2, 1}, pdigitback[-6712], pdigitback[x], pdigitback[{4, 3, 2}], {3},
 {6, 9, 2, 7, 6, 9, 4, 9, 2, 4}, pdigitback[22.], pdigitback[0], pdigitback[2, 2], pdigitback[]}
```

▹ **Condition** can go (almost) anywhere—even at the end of the whole definition.

For example, define different cases of the **check** function :

```
check[x_, y_] := Green /; x ≤ y
check[x_, y_] := Red /; x > y
{check[1, 2],
 check[2, 1],
 check[3, 4],
 check[50, 60],
 check[60, 50]}
```

{■, ■, ■, ■, ■}

```
? check
```

Symbol

Global`check

Definitions

check[x_, y_] := Green /; x ≤ y

check[x_, y_] := Red /; x > y

Full Name Global`check

▹ Double blank __ stands for any sequence of one or more arguments.

▹ Triple blank ___ stands for zero or more.

▸ For example, define a function that looks for Black and White (in this order) in a List, and picks out the shortest sequence between the first-met Black and its first subsequent White .

▹ Pattern { ___ , Black , m___ , White , ___ } matches Black followed by White, with any elements before/between/after them:

```
bw[ {__, Black, m__, White, __} ] := {m};
(* testcolor={RGBColor[0.5,0.5,0.5],
    GrayLevel[1], GrayLevel[0], RGBColor[0,0,1],
    GrayLevel[0], RGBColor[0,1,0], RGBColor[1,1,0],
    GrayLevel[1], RGBColor[1,0,1], GrayLevel[1],
    GrayLevel[0], RGBColor[0.5,0.5,0.5], GrayLevel[1],
    GrayLevel[0], RGBColor[0,0.5,0], RGBColor[0.5,0.5,0.5],
    GrayLevel[1], RGBColor[0.5,0,0], GrayLevel[1]}; *)
testcolors = {Gray, White, Black, Blue, Black, Green, Yellow, White,
    Purple, White, Black, Gray, White, Black, Green, Gray, White, Brown, White};
testcolors
bw[testcolors]
```

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

{■, ■, ■, ■}

▸ **bwex** picks the **Longest** sequence :

```
bwex[{__, Black, Longest[m__], White, __}] := {m};
testcolors
bwex[testcolors]
```

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

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

▸ **bwcut** cuts out the longest run containing only Black and White, and returns everything before the first Black/White and everything after the last Black/White .

x | y | z matches x or y or z

x .. matches any number of repetitions of x

```
bwcut[ {a__, Longest[(Black | White) ..], b__} ] := {{a}, Red, {b}};
(* testcolor2={RGBColor[0.5,0.5,0.5], RGBColor[0.5,0.5,0.5],
    GrayLevel[0], GrayLevel[1], GrayLevel[1], GrayLevel[0],
    GrayLevel[0], RGBColor[1,1,0]}; *)
testcolors2 = {Gray, Gray, Black, White, White, Black, Black, Yellow}
bwcut[testcolors2]
```

{■, ■, ■, □, □, ■, ■, ■}

{{■, ■}, ■, {■}}

▹ The pattern **x_** is actually short for

x : _

which means "match anything (i.e. **_**) and name the result **x** "

You can use notations like **x :** for more complicated patterns too .

‣ Define a pattern, named **m**, that matches a 2x2 matrix :

```
Clear[a, b];
grid22[ m : {{_, _}, {_, _}} ] := Grid[m, Frame → All];
test = { {{a, b}, {c, d}},
         {{12, 34}, {56, 78}} ,
         {123, 456} ,
         {{1, 2, 3}, {4, 5, 6}} };
Map[grid22, test]
```

a	b
c	d

12	34
56	78

, grid22[{123, 456}], grid22[{{1, 2, 3}, {4, 5, 6}}]}

‣ Name the sequence of Black and White, so it can be used in the result :

```
bwcut2[{a___, r : Longest[(Black | White) ..], b___}] := {{a}, Framed[Length[{r}]], {b}};
(* r  is  the longest sequence of Black(s) and White(s) *)
testcolors2 = {Gray, Gray, Black, White, White, Black, Black, Yellow}
bwcut2[testcolors2]
```

{■, ■, ■, □, □, ■, ■, ■}

{{■, ■}, 5, {■}}

```
(*  testcolors3={Green,White,White,White,White,White,White,
  White,Blue,Black,Gray,Gray,Black,White,White,Black,Black,Yellow,
  White,White,Black,Red,White,Black,White,White,White,Black,Black};
bwcut2[testcolors3] *)
```

▹ Let us use patterns to implement the classic computer science (Bubble Sort) **algorithm,** for **sorting a list**, by repeatedly swapping pairs of successive elements that are found to be out of order. We write each step in the algorithm as a replacement for a pattern.

```
(*    Swap  the  first pair  of  elements  found  to  be  out of order  *)
testlist0 = {5, 4, 1, 3, 2};
pattern = {x___, b_, a_, y___} /; b > a → {x, a, b, y};
ReplaceAll[testlist0, pattern]
```

{4, 5, 1, 3, 2}

```
(*  Do the same operation 8 times,
eventually sorting this particular list completely   *)
testlist = {4, 5, 1, 3, 2};
pattern = {x__, b_, a_, y__} /; b > a → {x, a, b, y};
NestList[
 # /. pattern &, (*  ReplaceAll[#, pattern] & , *)
 testlist,
 8]
```

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

▹ We do not know how long this method will take to finish sorting a particular list.
Therefore, use **FixedPointList**, which is like **NestList**, except that you do not have to tell a specific number of steps: it just goes on until the result reaches a fixed point, where nothing more is changing.

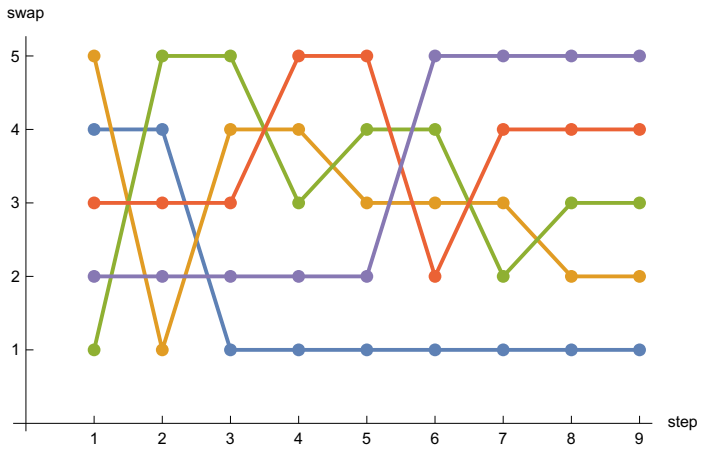
```
(*   Do the operation until a fixed point is reached *)
testlist = {4, 5, 1, 3, 2};
pattern = {x__, b_, a_, y__} /; b > a → {x, a, b, y};
fpl = FixedPointList[
   # /. pattern &, (*  ReplaceAll[#, pattern] & , *)
   testlist];
fpl;
tfpl = Transpose[fpl];
TableForm[tfpl]
```

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

Use **Transpose**, to find the current positions of each element (to be sorted) at each of the step taken.
ListPlot shows how the sorting process proceeds:
from step 1 to step 2, the swap $4 \longleftrightarrow 5$ occurs;
from step 2 to step 3, the swap $1 \longleftrightarrow 5$ occurs;
and so on.
From step 8 to step 9, no swap occurs, therefore the fixed point is reached.

```
(* fpl is the output List, containing all swaps performed to reach the FixedPoint*)
dims = Dimensions[fpl];
tfpl = Transpose[fpl];
TableForm[tfpl]
points = ListPlot[tfpl, PlotStyle → PointSize[0.02]];
lines = ListLinePlot[tfpl];
Show[points, lines, Ticks → Range[dims], AxesLabel → {"step", "swap"}]
(* X axis shows step K *)
(* Y axis shows the swap occurred at iteration K *)
(* At step 1→2, swap 1↔5 occurs *)
(* At step 2→3, swap 1↔4 occurs *)
(* At step 3→4, swap 3↔5 occurs *)
(* At step 4→5, swap 3↔4 occurs *)
(* At step 5→6, swap 2↔5 occurs *)
(* At step 6→7, swap 2↔4 occurs *)
(* At step 7→8, swap 2↔3 occurs *)
```

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

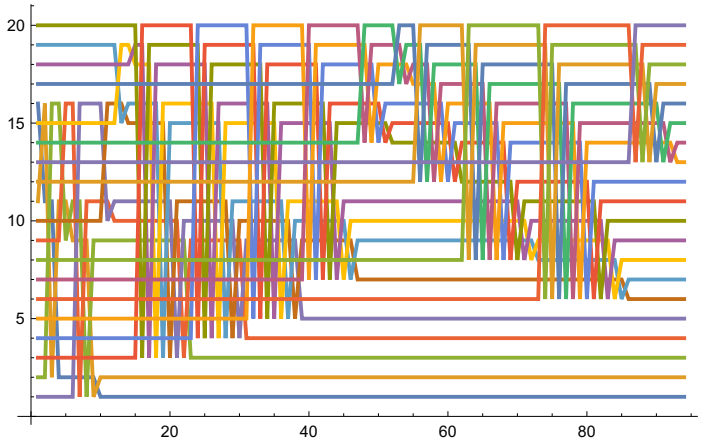


Sort a list of 20 elements obtained by randomly sampling Range[20].

```
SeedRandom[3];
testlistR = RandomSample[Range[20]]
pattern = {x___, b_, a_, y___} /; b > a → {x, a, b, y}; fplR = FixedPointList[
  # /. pattern &, (*  ReplaceAll[#, pattern] & , *)
  testlistR];
dimsR = Dimensions[fplR]
tfplR = Transpose[fplR];
linesR = ListPlot[tfplR,  Joined → True]
```

```
{16, 11, 2, 9, 1, 10, 19, 15, 18, 20, 3, 4, 5, 7, 14, 17, 12, 8, 6, 13}
```

```
{94, 20}
```



Vocabulary

patt**;**cond	a pattern that matches if a condition is met
___	a pattern for any sequence of zero or more elements (“triple blank”)
patt..	a pattern for one or more repeats of patt
Longest[patt]	a pattern that picks out the longest sequence that matches
Shortest[patt]	a pattern that picks out the shortest sequence that matches
FixedPointList[f,x]	keep nesting f until the result no longer changes

Exercises

Exercises

41.1 In the Squares of the first 50 Integers, find those containing successive repeated Digits.

41.2 In the first 100 Roman Numerals, find those containing L, I, X, in this order.

41.3 Define a function **g** that tests whether a non-empty List of Integers is the same as its Reverse.

41.4 Obtain a List of pairs of successive words, in the Wikipedia article on alliteration, that have identical first letters and such a first letter is a vowel: show its first, penultimate and 20th elements .

41.5 Use Grid to show the sorting process, seen in this section for {4, 5, 1, 3, 2}, with successive steps going down the page.

41.6 Use ArrayPlot to show the sorting process, seen in this section for a list of length 20, with successive steps going across the page.

41.7 Start with 1.0, then repeatedly apply (# + 2 / #) / 2 & until the result no longer changes.

41.9 Define Combinators, using the Rules s[x_][y_][z_]→x[z][y[z]], k[x_][y_]→x and, then, starting with s[s][k][s[s[s]][s]][s] , generate all possible combinations, applying the Rules until nothing changes. Give the last result.

41.10 Remove all trailing 0's from the Digit List for Factorial[20].

Q & A

Tech Notes

More to Explore

Guide to Patterns in *Mathematica*.

```
Hyperlink["https://reference.wolfram.com/language/guide/Patterns.html"]
```

https://reference.wolfram.com/language/guide/Patterns.html