

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 varietà di funzioni built-in, per alterare la struttura di espressioni.

Ad esempio, una regola di sostituzione (Replacement Rule) puo' avere un pattern nella sua componente sinistra (left-hand side): **lhs** \rightarrow 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]]]
```

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

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

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

```
Power2[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[Power2[4, 2] == Power[4, 2]2]
```

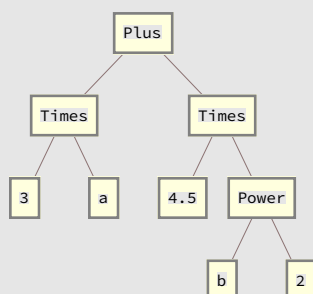
```
False
```

```
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 b2;
```

```
TreeForm[expr, ImageSize → Small]
```



```
(* match con la Head Plus : sostituisco Plus con Times *)
```

```
(* expr = 3 a + 4.5 b2 *)
```

```
expr /. Plus → Times
```

```
ReplaceAll[expr, Plus → Times];
```

```
13.5 a b2
```

```
(* nessuna sostituzione, perche' x non e' presente in expr *)
```

```
(* expr = 3 a + 4.5 b2 *)
```

```
expr /. x → x2
```

```
3 a + 4.5 b2
```

```
(* match con a : sostituisco a con x2 *)
```

```
(* expr = 3 a + 4.5 b2 *)
```

```
expr /. a → x2
```

```
4.5 b2 + 3 x2
```

```
(* 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 =  $\frac{f[a] + g[b]}{f[a, b]}$ ;
```

⌘ 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' $\text{expr_f} \rightarrow \text{expr}^2$.

Ne segue che ad f[a] viene sostituito f[a]^2 .

Analogamente f[a,b] viene rimpiazzato con f[a, b]^2

```

test =  $\frac{f[a] + g[b]}{f[a, b]}$ ;
test /. expr_f  $\rightarrow$  expr^2
(* Possiamo scrivere anche come segue: *)
test /. t_f  $\rightarrow$  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 è **f[x_] \rightarrow x^2**.

Ne segue che **f[a]** viene sostituito con **a^2**.

```

test =  $\frac{f[a] + g[b]}{f[a, b]}$ ;
test /. f[x_]  $\rightarrow$  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 è **f[x_, y_] \rightarrow (x+y)^2**.

Ne segue che **f[a,b]** è sostituito da **(a+b)^2**.

```

test =  $\frac{f[a] + g[b]}{f[a, b]}$ ;
test /. f[x_, y_]  $\rightarrow$  (x + y)^2

```

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

```

(* Esempio 3bis *)
test =  $\frac{f[a] + g[b]}{f[a, b]}$ ;
test /. f[x_, y_]  $\rightarrow$  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 è f[x_,y_]  $\rightarrow$  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 =  $\frac{x}{\text{Exp}[y]}$ ;
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 $\text{Exp}[t_] \rightarrow t$

Se l'intento fosse stato quello di ottenere $\frac{x}{y}$, resteremmo sorpresi dal risultato della sostituzione:

```
test =  $\frac{x}{\text{Exp}[y]}$ ;
test /. Exp[t_] -> t
-x y
```

Capiamo il motivo del risultato della sostituzione, esaminando la forma interna (FullForm) di **test** 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, et}
```

La sostituzione e' fatta col pattern **Power[E, Pattern[t, Blank[]]]** seguendo la regola **Power[E, Pattern[t, Blank[]]] \rightarrow t**

Qui il match e' $\text{Power}[E, \text{Times}[-1, y]] \rightarrow \text{Times}[-1, y]$
 ossia $E^{-y} \rightarrow -y$ ovvero $\text{Exp}[-y] \rightarrow -y$

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

```
{e-y, -y}
```

Dunque otteniamo:

```
test =  $\frac{x}{\text{Exp}[y]}$ 
test /. Exp[t_] -> t
(* Times e' n-aria *)
test /. Exp[t_] -> t // FullForm ;
```

```
e-y x
```

```
-x y
```

Se l'intento fosse stato quello di ottenere $\frac{x}{y}$, avremmo potuto definire il pattern seguente:

```
test /. Exp[t_] -> -1 / t
```

```
 $\frac{x}{y}$ 
```

Alternativa. Per ottenere $\frac{x}{y}$, avremmo potuto definire il pattern seguente:

```
test /. 1 / Exp[t_] -> 1 / t
```

```
 $\frac{x}{y}$ 
```

Programmazione basata su Regole

■ 6.1. Pattern *

□ 6.1.2 De-strutturare

Un pattern puo' essere costruito da qualsiasi espressione semplicemente sostituendo Blank con varie 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, {{i1, j1, ...}, {i2, j2, ...}, ...}] 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
```

```
( x1 Log[y1] )
( x2 Log[y2] )
( x3 Log[y3] )
( x4 Log[y4] )
```

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]}}
```

```
( x1 Log[y1] )
( x2 Log[y2] )
( x3 Log[y3] )
( x4 Log[y4] )
```

□ **Esercizio 1 pagina 144**

Che succede nei casi seguenti?

```
(* NO *)
dataDue = {{x1, y1}, {x2, y2}}
(* La lista dataDue ha esattamente 2 sottoparti .
   x_ intercetta la parte 1 di dataDueBis;
   y_ intercetta la parte 2 di dataDue *)
dataDue /. {x_, y_} -> {x, Log[y]}
```

```
{{x1, y1}, {x2, y2}}
```

```
{{x1, y1}, {Log[x2], Log[y2]}}
```

```
(* NO *)
dataDueBis = {{x1, y1, z1}, {x2, y2, z2}}
(* La lista dataDueBis ha esattamente 2 sottoparti .
   x_ intercetta la parte 1 di dataDueBis;
   y_ intercetta la parte 2 di dataDueBis *)
dataDueBis /. {x_, y_} -> {x, Log[y]}
```

```
{{x1, y1, z1}, {x2, y2, z2}}
```

```
{{x1, y1, z1}, {Log[x2], Log[y2], Log[z2]}}
```

```
(* OK *)
dataUno = {{x1, y1}}
dataUno /. {x_, y_} -> {x, Log[y]}
```

```
{{x1, y1}}
```

```
{{x1, Log[y1]}}
```

```
(* OK *)
dataTre = {{x1, y1}, {x2, y2}, {x3, y3}}
dataTre /. {x_, y_} -> {x, Log[y]}
```

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

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

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}
```

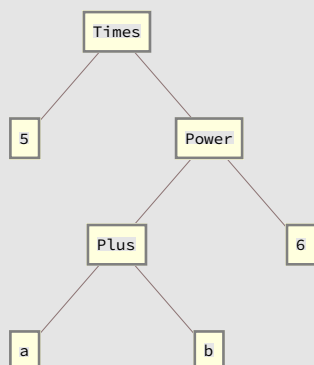
```
{b}
```

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 exprNonLista e' 5.

Gli interi da livello 1 e fino al livello 2 in exprNonLista 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 $1 - \text{value}$ for each RGB component. If you negate Red, Green, Blue, you get Cyan, Magenta, Yellow.

```
(* Red==RGBColor[1,0,0], Cyan==RGBColor[0,1,1] *)
```



```
{Red, Cyan, ColorNegate[Cyan] == Red}
```



```
(* Green==RGBColor[0,1,0],Magenta==RGBColor[1,0,1] *)
```

```
{Green, Magenta, ColorNegate[Magenta] == Green}
```

```
(* Blue==RGBColor[0,0,1],Yellow==RGBColor[1,1,0] *)
```

```
{Blue, Yellow, ColorNegate[Yellow] == Blue}
```

```
{, , True}
```

```
{, , True}
```

```
{, , True}
```

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

```
Blend[{Yellow, Pink, Green}]
```

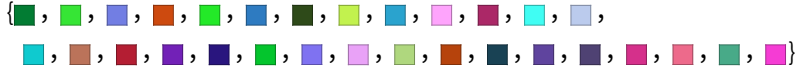
```
(* Mescola = Blend needs 1 compulsory argument *)
```

```
(* NO: Blend[Yellow,Pink,Green] *)
```

```
(* NO: Blend[] *)
```





```
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, X, X, X}
```

```
(* Number 100 in different sizes*)
Table[ Style[100, n], {n, 8, 30, 2} ]

{100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 100, 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, ...

`RGBColor[0.4,0.7,0.3]`

`Hue[0.8]`

`RandomColor[]`

`ColorNegate[Red]`

`Blend[{Red,Blue}]`

`Style[x,Red]`

`Style[x,20]`

`Style[x,20,Red]`

\$FontFamilies

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

Manipulate

GrayLevel

MapThread

Cases, Except, Alternatives

colors

red, green, blue color

color specified by hue

randomly chosen color

negate a color (complement)

blend a list of colors

style with a color

style with a size

style with a size

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 $\omega = 1$.

```
{LABColor[1, -1, 1],
  LABColor[1, 1, -1],
  LABColor[1, 0, -1],
  LABColor[1, 0, 1]}
```

{, , , 

`XYZColor[x, y, z]` , where **x** is color (combination of Red/Green) ,



y is Luminance (visually perceived brightness) , **z** is color (Blue) .

```
{XYZColor[0, 1, 0],
  XYZColor[1, 1/2, 0],
  XYZColor[1, 1, 0],
  XYZColor[0, 0, 1]}
```

{, , , 

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

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

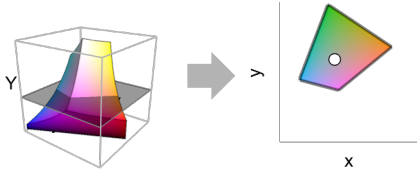
{, True, , True}

ChromaticityPlot and ChromaticityPlot3D plot lists of colors in color space.

ChromaticityPlot is used to visualize one or several colors in an image.

`ChromaticityPlot[colspace]` plots a 2D slice of the color space *colspace*

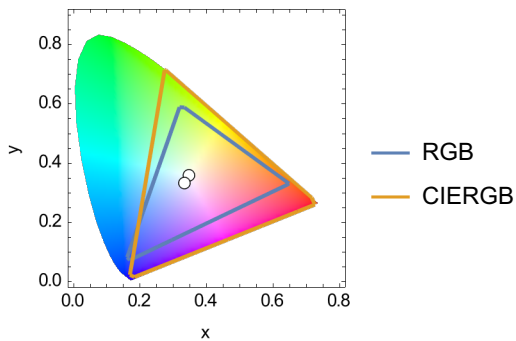
(i.e., convert the color coordinates in *colspace* to coordinates in *refcolspace* color space, and displays a slice given by constant luminance 0.01).



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

```
(* Confronto la gamma dei due spazi colore RGB e CIERGB
(CIE:Commission Internationale de l'Eclairage,
International Commission on Illumination) *)
```

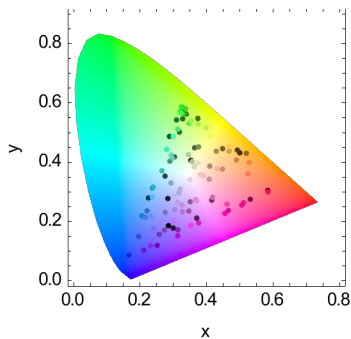
```
ChromaticityPlot[{"RGB", "CIERGB"}, ImageSize → Small]
```



```
(* Visualizzo una lista di colori RGB random *)
```

```
SeedRandom[8];
```

```
ChromaticityPlot[RandomColor[100], ImageSize → Small]
```



`ChromaticityPlot[image]` plots the pixels of *image* as individual colors.

→ It can be used to visualize the pixels of an image.

```
(* img=ExampleData[{"TestImage", "Peppers"}]; *)
```

```
img = ExampleData[{"TestImage", "Apples"}];
```

```
{Image[img, ImageSize → Small],
 ChromaticityPlot[img, ImageSize → Small]}
```

```
img = ExampleData[{"TestImage", "Tree"}];
```

```
{Image[img, ImageSize → Small],
 ChromaticityPlot[img, ImageSize → Small]}
```



```
(* img=ExampleData[{"TestImage","Bridge"}]; *)  
img = ExampleData[{"TestImage", "Moon"}];  
{Image[img, ImageSize → Small],  
ChromaticityPlot[img, ImageSize → Small]}
```

You can set lots of other style attributes in *Mathematica*, like **Bold**, **Italic** and **FontFamily**.

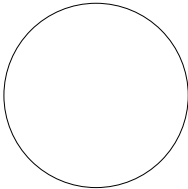
Basic Graphics Objects

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]
```

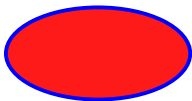


```
(* FullForm[g1]
FullForm[g2] *)
```

Use the graphics directive **EdgeForm** :

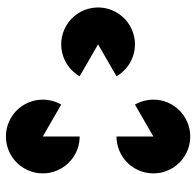
```
objEF = { EdgeForm[{Thick, Blue}],
  Opacity[0.9],
  Red,
  Disk[{0, 0}, {2, 1}] };
Graphics[objEF, ImageSize → Tiny]

(* 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 {tk-1, tk}
  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}, {rx,ry}, {tk-1,tk} *)
```

```

      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}}

```

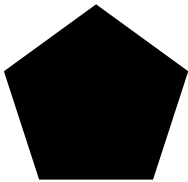
NOTE 1. Operators of (pre) increment / decrement

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]
```

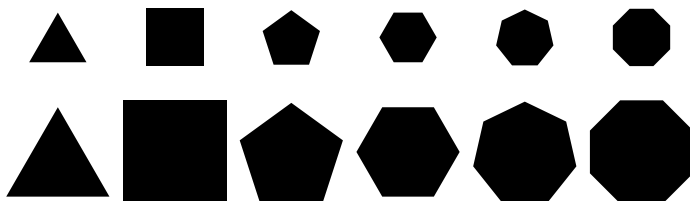


```
(* One Table with two iterators *)
```

```
grs = Table[
  Graphics[RegularPolygon[n], ImageSize → s],
  {s, {Tiny, Small}},
  {n, 3, 8}
];
```

```
(* TableForm[grs] *)
```

```
GraphicsGrid[ grs , ImageSize → Medium]
```

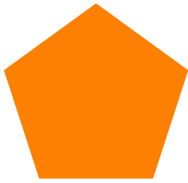


```
(* Another Table with the two iterators swapped *)
```

```
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];

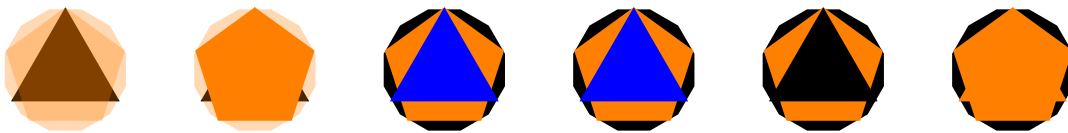
p3 = {dodeca,
      p0, (* {Orange,penta} *)
      Blue, tria};
g3 = Graphics[p3];
```

```
(* 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];

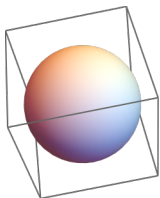
GraphicsRow[{g1, g2, g3, g4, g5, g6}, Spacings -> 50]
```



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).

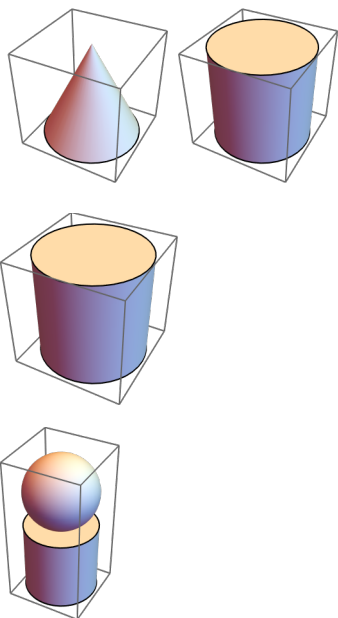

```

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]

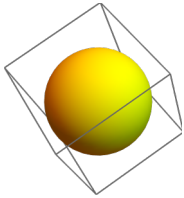
(* Sfera esterna al cilindro *)
Graphics3D[
  { Sphere[{0, 0, 2}], Cylinder[] },
  ImageSize -> Tiny]

```



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]
```



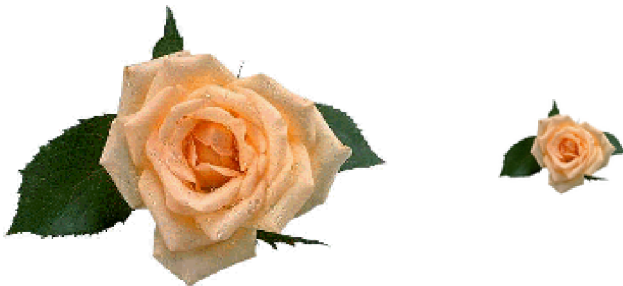
Have a look at the Help Page of **ImageSize**.

- 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` (panello), 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

<code>Circle[]</code>	specify a circle
<code>Disk[]</code>	specify a filled-in disk
<code>RegularPolygon[n]</code>	specify a regular polygon with n sides
<code>Graphics[object]</code>	display an object as graphics
<code>Sphere[], Cylinder[], Cone[], ...</code>	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>