1.2 Ingredienti di base per un Pacchetto

Lo scopo di pacchetto è quello di fare in modo che le funzioni (che lo compongono) si comportino come le Built-in.

- Esse devono essere documentate (anche in modo da potere usare la Query ?) .
- Il loro comportamento non deve dipendere da calcoli precedenti, svolti nella sessione corrente di Mathematica, prima di caricare il pacchetto.

- **Come (e perché) scrivere un pacchetto.**

Molte sono le cose che possono andare storte:

- potremmo avere valori di variabili (usate nelle funzioni del pacchetto) già definite nella sessione;
- potremmo passare (come argomenti) variabili che sono usate anche localmente dentro le funzioni del pacchetto;
- potremmo avere dato a una funzione del pacchetto lo stesso nome di un'altra funzione già definita da qualche altra parte (shadowing);
- funzioni ausiliarie e variabili private (che sono usate dentro il pacchetto) potrebbero essere accessibili all' utente.

- **LINKS to Modularity and the Naming of Things**

  - Modules and Local Variables
  - Local Constants
  - How Modules Work
  - Variables in Pure Functions and Rules
  - Variables in Mathematics
  - Blocks and Local Values
  - Blocks Compared with Modules
  - Contexts

  ⟷ Contexts and Packages

  - Wolfram Language Packages

  ⟷ Setting Up Wolfram Language Packages

  - Files for Packages
  - Automatic Loading of Packages
  - Manipulating Symbols and Contexts by Name
  - Intercepting the Creation of New Symbols

```
Hyperlink [
 "https://reference.wolfram.com/language/tutorial/ModularityAndTheNamingOfThings.html#3434"]
```

```
https://reference.wolfram.com/language/tutorial/ModularityAndTheNamingOfThings.html#3434
```

- **LINKS to Functions and Program**

  - Defining Functions
  - Functions as Procedures

  ⟷ Manipulating Options

  - Repetitive Operations
  - Transformation Rules for Functions

```
Hyperlink ["https://reference.wolfram.com/language/tutorial/FunctionsAndPrograms.html#15639"]
```

```
https://reference.wolfram.com/language/tutorial/FunctionsAndPrograms.html#15639
```

- **LINKS to Patterns**

- Introduction to Patterns
- Finding Expressions That Match a Pattern
- Naming Pieces of Patterns
- Specifying Types of Expression in Patterns
- Putting Constraints on Patterns
- Patterns Involving Alternatives
- Pattern Sequences
- Flat and Orderless Functions
- Functions with Variable Numbers of Arguments

⟷ Optional and Default Arguments

⟷ Setting Up Functions with Optional Arguments

- Repeated Patterns
- Verbatim Patterns
- Patterns for Some Common Types of Expression
- An Example: Defining Your Own Integration Function

```
Hyperlink["https://reference.wolfram.com/language/tutorial/Patterns.html#14984"]
```

```
https://reference.wolfram.com/language/tutorial/Patterns.html#14984
```

Prima di vedere, per passi, un esempio di creazione di un pacchetto, **Settiamo la Directory** in cui svilupperemo il codice.

```
(* Chiediamo quale è, correntemente , la Directory di lavoro *)
currWorkDir = Directory[];
(* Chiediamo quale è la Directory di questo stesso notebook *)
noteDir = NotebookDirectory[];
(* Chiediamo quali sono i Path noti al Kernel *)
(* currWorkDir è già in $Path *)
(* noteDir non è ancora in $Path *)
$Path;
(* Part[$Path,10] *)
(* Modo 1. Inseriamo noteDir in $Path *)
(* $Path=Append[$Path, noteDir] *)
AppendTo[$Path, noteDir];
(* Modo 2. Usiamo SetDirectory in modo che
 noteDir diventi la Directory corrente di lavoro *)
SetDirectory[noteDir];
(* Ora la Directory corrente di lavoro è noteDir *)
Directory[] == SetDirectory[noteDir]
```

```
True
```

1.2.0 Esempio di cattivo stile di programmazione.

Consideriamo il file **BadExample.m**, che contiene la seguente riga di codice:

```
PowerSum[x_, n_] := Sum[x^i, {i, 1, n}]
```

Carichiamo BadExample.m

```
(* Get reads in a file, evaluating each expression in it,
and returning the last one *)
AppendTo[ $Path , NotebookDirectory []];
Get["BadExample .m"]
(* or Get["BadExample` "] *)
(* or <<BadExample .m *)
(* or <<BadExample` *)

(* After setting up the Directory , FindFile helps in finding a file *)
(* ff=FindFile["BadExample` "]; Get[ff] *)
```

Fin qui, nessun problema; con una chiamata a **PowerSum[a , 5]**
oppure a **PowerSum[x , 5]** otteniamo il risultato voluto:

```
PowerSum[a, 5]
```

$$a + a^2 + a^3 + a^4 + a^5$$

```
PowerSum[x, 5]
```

$$x + x^2 + x^3 + x^4 + x^5$$

Qui, invece, sorgono problemi.
Con una chiamata a **PowerSum[i , 5]** non otteniamo il risultato voluto.
La variabile **i** viene "catturata" dalla variabile omonima ,
usata (nel range { **i , 1 , n** } della Sommatoria **Sum**)
dalla funzione **PowerSum** del pacchetto **BadExample.m**

Cosi', invece di ottenere $i + i^2 + i^3 + i^4 + i^5$,
otteniamo un numero $(1 + 2^2 + 3^3 + 4^4 + 5^5)$:

```
PowerSum[i, 5]
```

```
3413
```

```
PowerSum[z, 4]
```

$$z + z^2 + z^3 + z^4$$

```
? Global`*
```

```
Global`
  a          i          n          PowerSum    x          z
```

1.2.1 Isoliamo le variabili locali (con Module)

Isoliamo la variabile locale **i** usata nella Sommatoria dentro il pacchetto.

Per farlo, usiamo **Module**.

In questo modo, il simbolo locale usato è sempre nuovo, e non può entrare in conflitto con nulla che venga passato come parametro (argomento) alla funzione PowerSum del pacchetto.

Consideriamo il file **BetterExample.m**, che contiene il seguente codice:

```
PowerSum[ x_ , n_ ] := Module[{ i }, Sum[ x^i , { i , 1 , n } ] ]
```

```
(* Prima di caricare il pacchetto , quittare il kernel *)
(* ClearAll["Global`*"]; *)
AppendTo[ $Path , NotebookDirectory []];
<< BetterExample.m
```

Nessun problema qui :

```
PowerSum[a, 5]
```

$$a + a^2 + a^3 + a^4 + a^5$$

Nessun problema anche qui:

```
PowerSum[i, 5]
```

$$i + i^2 + i^3 + i^4 + i^5$$

Resta però un problema :
i simboli ausiliari **i , n , x** (usati nel pacchetto BetterExample.m)
sono visibili anche nel contesto **Global** (fuori dal pacchetto):

```
(* notiamo la variabile "i", usata in questo notebook nella chiamata di PowerSum[i,5],
e la sua copia locale "i$" usata nel pacchetto BetterExample .m *)
? Global`*
```

```
Global`
  a              i              i$             n              PowerSum       x
```

1.2.2 Mettere le cose nel loro Contesto corretto (Private)

Il contesto **Context** è il meccanismo che *Mathematica* mette a disposizione,
per mantenere differenti :
- le variabili usate in un pacchetto,
- dalle variabili usate nella sessione principale .

Ogni simbolo appartiene ad un determinato Contesto.
All' interno di un unico Contesto, i nomi dei simboli sono univoci.
Uno stesso nome, invece, può apparire in due Contesti differenti.
Di default, tutti i nuovi simboli che noi definiamo vengono messi nel contesto Global`.

In **BetterExample.m** , perfino la variabile **i** locale (**i$**) entra nel contesto Global` .
Per evitare ciò, dobbiamo dire a *Mathematica* di creare nuovi simboli in un contesto differente.

Consideriamo il file **BestExample.m**, che contiene il seguente codice:

```
PowerSum::usage = "PowerSum[ x , n ] returns the sum of the first n powers of x."

Begin["Private`"]
PowerSum[ x_ , n_ ] := Module[{ i } , Sum[ x^i , { i , 1 , n } ] ]
End[ ]
```

```
(* Prima di caricare il pacchetto , quittare il kernel *)
(* ClearAll["Global`*"]; *)
AppendTo[ $Path , NotebookDirectory []];
<< BestExample.m ;
```

Questa volta, la variabile **i** locale (**i$**) viene creata dentro al contesto Private` ,

che **non** verrà esaminato quando (successivamente) noi useremo nomi di variabile nella sessione corrente.

Il simbolo **PowerSum** (che è il nome della funzione che vogliamo usare, dopo avere caricato il pacchetto che la contiene), invece, deve ovviamente essere visibile nel contesto Global` .

Nel pacchetto **BestExample.m** inseriamo, fuori del contesto Private`, la riga di documentazione (usage) della funzione **PowerSum .**

```
PowerSum[a, 5]
PowerSum[i, 5]
```

$a + a^2 + a^3 + a^4 + a^5$

$i + i^2 + i^3 + i^4 + i^5$

```
? Global`*
```

Global`

a i PowerSum

```
? Private`*
```

Private`

i i$ n x

```
? PowerSum
```

Symbol

PowerSum [x, n] returns the sum of the first n powers of x.

1.2.3 Il contesto di un pacchetto

Oltre a nascondere le variabili locali e le funzioni ausiliarie (in modo che non siano visibili nel contesto Globale) ,
vogliamo mettere tutte le funzioni del pacchetto in un unico contesto separato (e **con un proprio nome**, che non sia il generico Private`)

Tale **nome** (da noi scelto per il contesto del pacchetto), però, deve essere visibile (altrimenti non potremmo usarne le funzionalità interne).
Questo si ottiene con **BeginPackage[ ] / EndPackage[ ]**

Consideriamo un nuovo pacchetto esemplificatore : **MappaCartesiana.m**
che contiene il codice che segue , e che crea il grafico di una griglia, del piano cartesiano,
in cui i vertici sono punti { x , y } trasformati da una funzione **f**

Nota.
Prima vengono formati i valori **f**[x + I* y];
poi ciascun valore viene separato nelle sue parti Reale e Immaginaria .
La tabella di coppie {Re , Im} viene infine graficata in forma parametrica:
- la tabella dei valori orizzontali (spread = { x , x0 , x1 , dx }) risulta dipendente solo da **y** ,

i.e. in tale tabella coppie sono di tipo { **Re[y] , Im[y]** } (a meno della funzione **f**);

pertanto, su tale tabella, ParametricPlot ha bounds={ y , y0 , y1 };
▫ la tabella dei valori verticali (spread = { y , y0 , y1 , dy }) risulta dipendente solo da **x** ,
i.e. in tale tabella coppie sono di tipo { **Re[x]** , **Im[x]** } (a meno della funzione **f**);
pertanto, su tale tabella, ParametricPlot ha bounds={ x , x0 , x1 }.

CODICE di MappaCartesiana .m

```
BeginPackage["MappaCartesiana`"]

CartesianMap::usage =
"CartesianMap[ f , { x0 , x1 , dx }, { y0 , y1 , dy } ] plots the image
of the Cartesian coordinate lines under the function f ."

Begin["`Private`"]

CartesianMap[ func_ , { x0_ , x1_ , dx_ }, { y0_ , y1_ , dy_ } ] :=
  Module[ { xy , x , y , horizontalgrid , verticalgrid },

    xy = func[x + I y];

    horizontalgrid = Curves[ xy , { x , x0 , x1 , dx }, { y , y0 , y1 } ];
    verticalgrid   = Curves[ xy , { y , y0 , y1 , dy }, { x , x0 , x1 } ];

    Show[
    Graphics[ Join[ horizontalgrid, verticalgrid ] ],
     AspectRatio → Automatic, Axes → True ]
 ]

Curves[ xy_ , spread_ , bounds_ ] :=
 Module[ { curves } ,
    curves = Table[ { Re[xy] , Im[xy] } , spread ] ;
    ParametricPlot[curves, bounds][[1]]
  ]

End[]
EndPackage[]
```

Notiamo che in **Begin**["`Private`"] c'è un doppio "tick".
Questo indica che il contesto **Private** è un sotto-contesto del contesto **MappaCartesiana`** del pacchetto .

```
AppendTo [ $Path , NotebookDirectory [] ];
<< MappaCartesiana.m
```

La funzione CartesianMap è nel proprio contesto MappaCartesiana`

```
Context [CartesianMap ]
```

```
MappaCartesiana`
```

Il contesto MappaCartesiana` è accessibile, perché è stato aggiunto al Path per la ricerca di Contesti :

```
$ContextPath
```

{MappaCartesiana` , WolframAppCatalog` , System` , Global` }

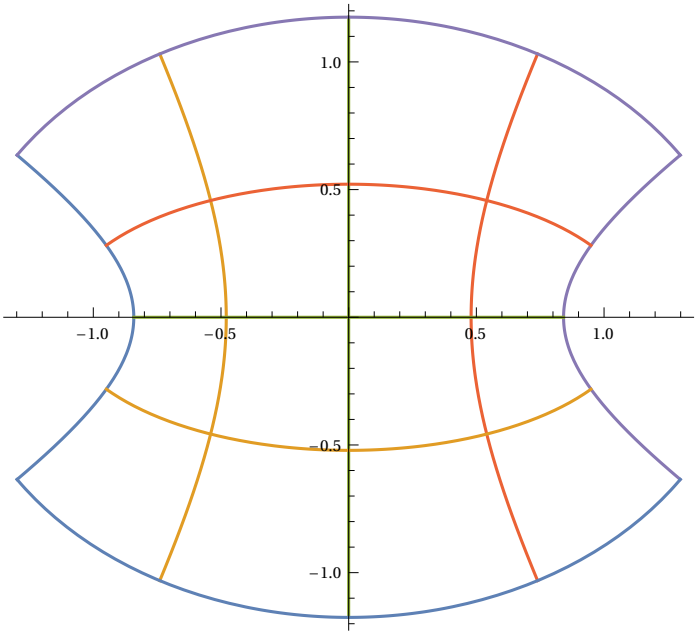
Chiamo la funzione:

```
? CartesianMap
```

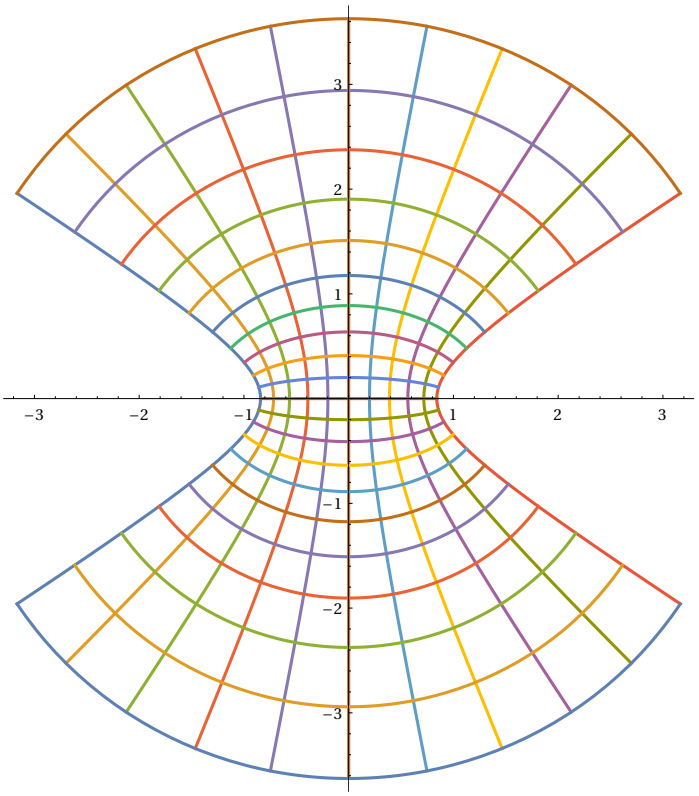
Symbol

CartesianMap [f , {x0, x1, dx}, {y0, y1, dy}] plots the image
of Cartesian coordinate lines, modified under the action of function f .

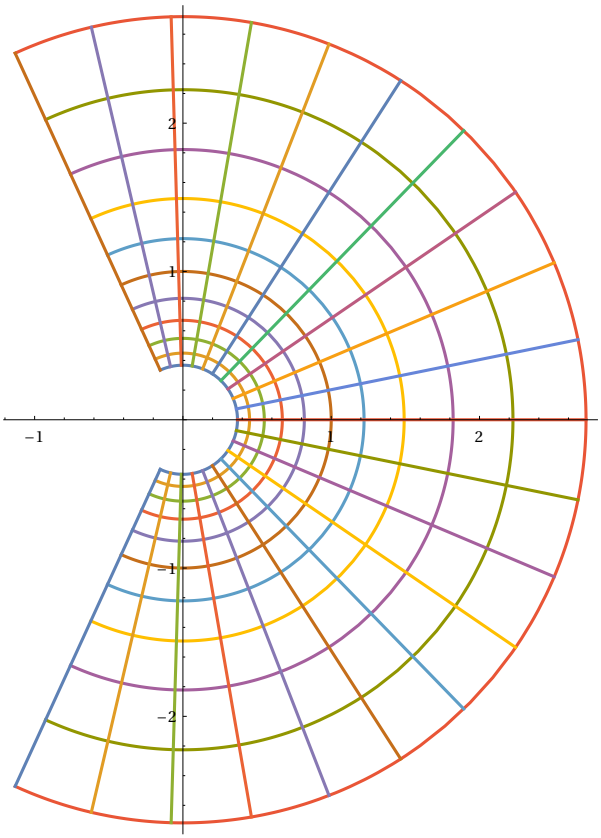
```
CartesianMap [Sin, {-1, 1, 0.5}, {-1, 1, 0.5}]
```



```
CartesianMap [Sin, {-1, 1, 0.2}, {-2, 2, 0.2}]
```



```
CartesianMap [Exp , {-1, 1, 0.2}, {-2, 2, 0.2}]
```




```
CartesianMap [Cos , {0.2 , Pi - 0.2 , (Pi - 0.4) / 19}, {-2 , 2 , 4 / 16}]
```