Wolfram

The Wolfram Language is the underlying language originally used in Mathematica, but now available for use in multiple contexts.

Wolfram Language has several interfaces: * The command line kernel interface on Raspberry Pi (just called *The Wolfram Language*) which runs interactively and can't produce graphical input. * *Mathematica* which is a rich text/maths editor with interactive Wolfram built in: pressing shift+Return on a "code cell" creates an output cell with the result, which is not dynamic * *Wolfram Workbench* which is Eclipse interfaced to the Wolfram Language backend

The code in this example can be typed in to any interface and edited with Wolfram Workbench. Loading directly into Mathematica may be awkward because the file contains no cell formatting information (which would make the file a huge mess to read as text) - it can be viewed/edited but may require some setting up.

```
(* This is a comment *)

(* In Mathematica instead of using these comments you can create a text cell
   and annotate your code with nicely typeset text and images *)

(* Typing an expression returns the result *)
2*2              (* 4 *)
5+8              (* 13 *)

(* Function Call *)
(* Note, function names (and everything else) are case sensitive *)
Sin[Pi/2]        (* 1 *)

(* Alternate Syntaxes for Function Call with one parameter *)
Sin@(Pi/2)       (* 1 *)
(Pi/2) // Sin    (* 1 *)

(* Every syntax in WL has some equivalent as a function call *)
Times[2, 2]      (* 4 *)
Plus[5, 8]       (* 13 *)

(* Using a variable for the first time defines it and makes it global *)
x = 5            (* 5 *)
x == 5           (* True, C-style assignment and equality testing *)
x                (* 5 *)
x = x + 5        (* 10 *)
x                (* 10 *)
Set[x, 20]       (* I wasn't kidding when I said EVERYTHING has a function equivalent *)
x                (* 20 *)

(* Because WL is based on a computer algebra system, *)
(* using undefined variables is fine, they just obstruct evaluation *)
cow + 5          (* 5 + cow, cow is undefined so can't evaluate further *)
cow + 5 + 10     (* 15 + cow, it'll evaluate what it can *)
%                (* 15 + cow, % fetches the last return *)
% - cow          (* 15, undefined variable cow cancelled out *)
moo = cow + 5    (* Beware, moo now holds an expression, not a number! *)

(* Defining a function *)
Double[x_] := x * 2    (* Note := to prevent immediate evaluation of the RHS
```

```
                        And _ after x to indicate no pattern matching constraints *)
Double[10]             (* 20 *)
Double[Sin[Pi/2]]      (* 2 *)
Double @ Sin @ (Pi/2)  (* 2, @-syntax avoids queues of close brackets *)
(Pi/2) // Sin // Double(* 2, //-syntax lists functions in execution order *)

(* For imperative-style programming use ; to separate statements *)
(* Discards any output from LHS and runs RHS *)
MyFirst[] := (Print@"Hello"; Print@"World")  (* Note outer parens are critical
                                                ;'s precedence is lower than := *)
MyFirst[]                                   (* Hello World *)

(* C-Style For Loop *)
PrintTo[x_] := For[y=0, y<x, y++, (Print[y])]  (* Start, test, incr, body *)
PrintTo[5]                                      (* 0 1 2 3 4 *)

(* While Loop *)
x = 0; While[x < 2, (Print@x; x++)]     (* While loop with test and body *)

(* If and conditionals *)
x = 8; If[x==8, Print@"Yes", Print@"No"]   (* Condition, true case, else case *)
Switch[x, 2, Print@"Two", 8, Print@"Yes"]  (* Value match style switch *)
Which[x==2, Print@"No", x==8, Print@"Yes"] (* Elif style switch *)

(* Variables other than parameters are global by default, even inside functions *)
y = 10             (* 10, global variable y *)
PrintTo[5]         (* 0 1 2 3 4 *)
y                  (* 5, global y clobbered by loop counter inside PrintTo *)
x = 20             (* 20, global variable x *)
PrintTo[5]         (* 0 1 2 3 4 *)
x                  (* 20, x in PrintTo is a parameter and automatically local *)

(* Local variables are declared using the Module metafunction *)
(* Version with local variable *)
BetterPrintTo[x_] := Module[{y}, (For[y=0, y<x, y++, (Print@y)])]
y = 20             (* Global variable y *)
BetterPrintTo[5]   (* 0 1 2 3 4 *)
y                  (* 20, that's better *)

(* Module actually lets us declare any scope we like *)
Module[{count}, count=0;        (* Declare scope of this variable count *)
  (IncCount[] := ++count);      (* These functions are inside that scope *)
  (DecCount[] := --count)]
count              (* count - global variable count is not defined *)
IncCount[]         (* 1, using the count variable inside the scope *)
IncCount[]         (* 2, incCount updates it *)
DecCount[]         (* 1, so does decCount *)
count              (* count - still no global variable by that name *)

(* Lists *)
myList = {1, 2, 3, 4}     (* {1, 2, 3, 4} *)
myList[[1]]               (* 1 - note list indexes start at 1, not 0 *)
Map[Double, myList]       (* {2, 4, 6, 8} - functional style list map function *)
Double /@ myList          (* {2, 4, 6, 8} - Abbreviated syntax for above *)
```

```
Scan[Print, myList]       (* 1 2 3 4 - imperative style loop over list *)
Fold[Plus, 0, myList]     (* 10 (0+1+2+3+4) *)
FoldList[Plus, 0, myList] (* {0, 1, 3, 6, 10} - fold storing intermediate results *)
Append[myList, 5]         (* {1, 2, 3, 4, 5} - note myList is not updated *)
Prepend[myList, 5]        (* {5, 1, 2, 3, 4} - add "myList = " if you want it to be *)
Join[myList, {3, 4}]      (* {1, 2, 3, 4, 3, 4} *)
myList[[2]] = 5          (* {1, 5, 3, 4} - this does update myList *)

(* Associations, aka Dictionaries/Hashes *)
myHash = <|"Green" -> 2, "Red" -> 1|>   (* Create an association *)
myHash[["Green"]]                       (* 2, use it *)
myHash[["Green"]] := 5                  (* 5, update it *)
myHash[["Puce"]] := 3.5                 (* 3.5, extend it *)
KeyDropFrom[myHash, "Green"]            (* Wipes out key Green *)
Keys[myHash]                            (* {Red} *)
Values[myHash]                          (* {1} *)

(* And you can't do any demo of Wolfram without showing this off *)
Manipulate[y^2, {y, 0, 20}] (* Return a reactive user interface that displays y^2
                               and allows y to be adjusted between 0-20 with a slider.
                               Only works on graphical frontends *)
```

Ready For More?

- Wolfram Language Documentation Center