Programming Updates in Maple 2026

Programming in Maple 2026 includes a range of updates aimed at making it easier to build, package, and maintain Maple code. These updates span core language features, performance improvements, and enhancements to everyday programming workflows.

Vector Database

- Maple 2026 adds a vector database, or vector store, in the VectorSearch package. A vector database is a data structure that solves the following problem: given a set S consisting of a large number of vectors, generally in a high-dimensional vector space, quickly find some of the elements of S that are closest to a given vector v. A demonstration follows.
- Load the package.

```
> with(VectorSearch):
```

- Create a database. It will hold vectors of dimension 1500.

```
> dim := 1500:
```

```
> idx := Create(dim);
```

$$idx := \text{"External/0x7066169c39f0"}$$

- Add 1000 random vectors to it.

```
> n_vectors := 1000:
```

```
> s := Statistics:-Sample(Normal(0, 1), [n_vectors, dim]);
```

$s :=$

	1	2	3	
1	−1.07242412799827	0.307853552460182	−0.0121756776556214	⋯
2	−0.329077870547065	−0.256909788403948	1.22087383259245	⋯
3	−0.617091936909790	0.817823121679866	−2.02885609419507	⋯
4	0.214466745245291	1.62282191048178	−0.0381927024017555	⋯
5	−0.0254281280423846	−0.306374292255348	−0.294296891229388	⋯
6	1.72882128417783	0.116667007938814	−0.292207209938442	⋯
7	−1.63485675434390	−0.609021515874302	−0.126362221226092	⋯
8	1.57117217175430	−0.529288518847664	−2.00767278179774	⋯
9	0.170358421410408	0.634550196450233	0.817882733890058	⋯
10	1.00647840875181	−0.0104845279200755	0.387263088798052	⋯
	⋮	⋮	⋮	

1000 × 1500 Matrix

```
> for i to n_vectors do
    Add(idx, String(i), s[i]);
  end do:
```

• Construct a vector that is nearer some 20 of these vectors.

```
> v := add(s[50*i] / evalf(sqrt(i)), i=1..20);
```

$v :=$ [2.80199212052753, 2.65341938236729, −1.96034932346277, −3.59861548084517, −0.423954207444625, −0.294811959515540, −2.24363078071995, 1.09381653509538, 1.76102976656496, −1.52180115027421, −0.823529681348780, 0.302321701549215, 0.294754256372467, −0.0388696271772355, 1.37867784451913, −0.679190947220155, 0.231559424536255, −0.367783505765180, 4.41212814151400, −0.940309098861477, 0.299673413554745, 0.688630112644355, 2.70208344786189, 0.543382064790183, −3.06404824488061, 3.63728681284688, 1.52547241680483, 3.11333209409634, 2.10460180418302, 1.47135642209683, 2.08004825684926, −0.283497154970959, 1.08637894528499, −1.54187828775464, −0.516356965247718, −0.903399653868279, −1.98551936841108, −1.98245045591192, 0.425508199724943, −2.25103883870368, 1.89180525419206, −3.23662810775293, −2.02565597407352, 1.21103018695867, 0.861278486146034, 0.807861909451035, −2.55283002177097, 1.00205273841363, 3.48572173575679, −1.59135685774751, 1.87656665642381, 0.650256298340633, −0.180575695860342, 2.83924549229265, −1.15189243929714, −1.57991618649932, −1.00245136186906, −1.12006796588021, −2.80060473822163, 4.88639787521985,

$-2.40804191947622, 0.657732897433597, 2.94847372785928, -1.85035847367257,$
$3.80034090784925, -0.882675105585865, 0.539211364103269, 1.42470619185993,$
$2.44668860219479, 0.453403295257993, 0.777792465019312, -2.21264411742940,$
$-2.71919941539893, 0.874379113127774, 1.71964156946528, -0.529818175118459,$
$1.22747028448148, 1.26678317653742, -0.746130157220754, -0.0451638254000707,$
$-0.904815070363422, -0.468453050176093, 0.521241683837650, -2.35713663268976,$
$1.12458557849786, -3.83212101697786, -0.405608572243448, 2.13359801533939,$
$1.60358749789330, 1.77499327351991, 3.40932674758357, -0.667273927791160,$
$-0.661795868531221, -3.19473174189689, 2.87960926241030, 1.66177587563902,$
$1.66884538574861, 1.53877828604196, 0.423824532749809, -0.504802968600733, \ldots,$
$\cdots$ 1400 row vector entries not shown]

• Find 30 of the closest vectors.

```
> close_vector_keys := Search(idx, v, 30);
```

close_vector_keys := ["50", "100", "150", "200", "400", "600", "450", "500", "650", "1000", "900", "750", "824", "850", "700", "908", "520", "42", "968", "598", "354", "491", "402", "353", "142", "121", "704", "544", "144", "305"]

• Find the distance between these vectors and v.

```
> close_vectors := map(u -> s[parse(u)], close_vector_keys):
```

```
> distances := map(u -> LinearAlgebra:-Norm(u - v, 2),
  close_vectors);
```

distances := [62.5438625889896613, 68.7414664023135202, 71.2356253465894582, 73.2243420759500339, 75.5869436557170644, 76.1038691132943228, 76.1740596206177827, 76.2946729657286369, 76.3151404172196663, 77.5168598353440785, 77.6464148153114593, 78.1611045558024387, 78.3444816688743515, 78.4465262611677900, 78.8140270932480576, 79.2780748709537875, 79.3372688361419023, 79.5386427455342044, 79.5616010676925214, 79.7679921109387351, 79.9136614229822015, 79.9295310020447403, 79.9438564013452861, 79.9879031554994526, 80.0234984790753145, 80.0306862648519939, 80.0805288517314580, 80.1226046806954741, 80.1233072668303521, 80.1880108029501599]

• Save the vector database to a file.

```
> path := FileTools:-JoinPath(["filename.vdb"], 'base' = 'tempdir')
  ;
```

path := "/tmp/mpldoc9/filename.vdb"

```
> Save(idx, path);
```

• Load the vector database from this file.

```
> idx2 := Load(path);
```

$$idx2 := \text{"External/0x706615bfba20"}$$

- Search near the same random vector.

```
> Search(idx2, v, 30);
```

["50", "100", "150", "200", "400", "600", "450", "500", "650", "1000", "900", "750", "824", "850", "700", "908", "520", "42", "968", "598", "354", "491", "402", "353", "142", "121", "704", "544", "144", "305"]

with(Units:-Simple) and the evalhf Hardware Floating-Point Subsystem

- A call to evalhf evaluates an expression to a numerical value using the floating-point hardware of the underlying system. The evaluation is done in double precision. The argument passed to evalhf must be an expression that evaluates to float[8] or complex[8] values as singletons or in an array, matrix, or vector. Procedures and functions calculated must be known to evalhf or operate under the constraints of dealing with the types described.
- In the past, loading the Units:-Simple package would prevent subsequently defined procedures from being executed under evalhf. This is because bindings for standard operators like plus, times, and exponent are overloaded with equivalent versions that know how to deal with units. In Maple 2026, evalhf will not give an error when encountering Units:-Simple operators, allowing you to properly compute under evalhf given numeric arguments even when the Units:-Simple package is loaded.

```
> with(Units:-Simple):
> proc_with_unit_bindings := proc(a,b) a*b; end:
> lprint(eval(proc_with_unit_bindings));
proc (a, b) Units:-Simple:-`*`(a,b) end proc
> proc_with_unit_bindings(2,3);
```

$$6$$

```
> evalhf(proc_with_unit_bindings(2,3));
```

$$6.$$

New Type

- Maple 2026 introduces a new type, **variable** which is a combination of type,name and not type,constant. This type is particularly useful in combination with indets or subsindets when extracting variables from an expression.

```
> restart;
```

```
> type( x, 'variable');
```

$$true$$

```
> type( Pi, 'variable');
```

$$false$$

```
> indets( exp(1)+y*sin(Pi*x), 'variable' );
```

$$\{x, y\}$$

Filename Generation

- A new command, FileTools:-SuggestName helps generate data file names when you want to keep the various iterations of computed data in the same directory through multiple runs.
- The **SuggestName("test.abc")** command checks the file system for a specified file. If the file does not exist, **SuggestName** returns the given string unchanged; otherwise, it will repeat the process for "test (1).abc", then "test (2).abc", etc. until it finds a name that does not exist in the given path.

```
> out := FileTools[SuggestName]( "datafile.csv" );
```

"datafile.csv"

```
> Export(out,[1,2,3]):
```

```
> out := FileTools[SuggestName](out);
```

"datafile (1).csv"

```
> Export(out,[4,5,6]):
```

```
> out := FileTools[SuggestName](out);
```

"datafile (2).csv"

New Package Property

- The PackageTools package provides tools for managing workbook-based packages. You can use it to distribute Maple packages stored in workbook files.
- If a user installs your package, they will get new packages in their menu for loading packages. In Maple 2026, we have introduced a new property understood by the installation infrastructure: by setting the **packagelistforloadmenu** property with the PackageTools:-SetProperty command, you can control which packages get added to this menu.

Describing Maple Types in English

- The convert/english command can create English language descriptions of a few types of objects, including types as recognized by the type command. We overhauled the code for doing this for Maple 2026. Here are a few examples:

```
> T1 := [float, nonposint];
```

$$T1 := [float, nonposint]$$

- Now $T1$ represents the type of lists with two entries, where the first is a floating point number and the second is a non-positive integer.

```
> type([2.5, -3], T1);
```

$$true$$

```
> convert(T1, english, type);
```

"a list containing a floating point number and a non-positive integer"

```
> T2 := set(T1);
```

$$T2 := set([float, nonposint])$$

- Now $T2$ represents the type of sets of expressions of type_T1_.

```
> type({[2.5, -3], [1.1, 0]}, T2);
```

$$true$$

```
> convert(T2, english, type);
```

"a set of lists containing a floating point number and a non-positive integer"

Flexible Parameter Type Checking, Coercion, and User-Friendly Messages

- In Maple, procedures can be declared with strict types on their parameters. The coerce mechanism was introduced in order to give an easy way for a procedure to automatically accept many data structures while only having to deal with one in your code. For example, coerce can be used to accept lists, sets, and Arrays, while the body of your code only deals with one of these. There are a few ways to do this with the coerce mechanism, the most explicit is as follows:

```
> MakeArray := proc( a )
      if a::Array then
          return a;
      elif a::{list,set} then
          return convert(a,Array);
      else
```

```
        error "Array expected";
    end if;
  end proc:
```

```
> WorkOnArray := proc( Data::coerce(MakeArray) )
    whattype(Data);
  end proc:
```

Whether you call the WorkOnArray procedure with a list, set or Array, the body of the procedure will only ever see an Array.

```
> WorkOnArray(Array([1,2,3]));
```

Array

```
> WorkOnArray([1,2,3]);
```

Array

```
> WorkOnArray({1,2,3});
```

Array

A difficulty in previous versions of Maple was that the error message was incomplete. New in Maple 2026, the exception message indicates both the name and position of the argument.

```
> WorkOnArrays := proc( Data1::coerce(MakeArray), Data2::coerce
  (MakeArray) )
    [whattype(Data1),whattype(Data2)];
  end proc:
  WorkOnArrays([1],1);
```

```
Error, (in WorkOnArrays) invalid input: verification of 2nd argument,
'Data2', failed: Array expected
```

This update opens the door to using coerce as a general type checking mechanism for procedures with complicated type specifications. For example, consider the font option for plots, which must be a list indicating the font [family,style,size], but style and size are optional. A partial implementation using coerce might look like the following:

```
> checkFont := proc( font )
    if not font::list then
        error "expecting font to be a list";
    elif not numelems(font) in {1,2,3} then
        error "expecting font to be a list of no more than 3
  elements";
    elif not font[1] in {Times, Courier, Helvetica, Symbol} then
        error "expecting the first element of the list, the font
```

```
  family, to be one of Times, Courier, Helvetica, or Symbol";
     end if;
     font;
  end proc:
```

```
> MyPlot := proc( titlefont::coerce(checkFont) )
      titlefont;
  end proc:
```

Now the error message can be very precise about what was incorrect in the argument passed by the user. Also, the checkFont method can be generic, used for the axesfont, captionfont, labelfont, etc.

```
> MyPlot( [NewTimes] );
```

```
Error, (in MyPlot) invalid input: verification of 1st argument,
'titlefont', failed: expecting the first element of the list, the font
family, to be one of Times, Courier, Helvetica, or Symbol
```

Further, because this is using the coerce mechanism, the checkFont procedure can be adjusted to do double-duty, by both checking the input and converting to a normal form.

```
> checkFont := proc( font )
     local family := Times;
     local style := roman;
     local size := 12;
     if not font::list then
         error "expecting font to be a list";
     elif not numelems(font) in {1,2,3} then
         error "expecting font to be a list of no more than 3
  elements";
     elif not font[1] in {Times, Courier, Helvetica, Symbol} then
         error "expecting the first element of the list, the font
  family, to be one of Times, Courier, Helvetica, or Symbol";
     end if;
     [family, style, size];
  end proc:
```

```
> MyPlot := proc( titlefont::coerce(checkFont) )
      titlefont;
  end proc:
```

```
> MyPlot( [Times] );
```

[*Times*, *roman*, 12]

String Continuation in Text Files

• Maple provides several ways for entering a long string in a .mpl code file.

You can put two string constants side by side and the parser will concatenate them:

```
> "a"  "b";
```

"ab"

When a string spans multiple lines, you can add a backslash to the end of the line:

```
> "a\
  b";
```

"ab"

If you want the newline as part of the string, you can use "\n" as a newline character:

```
> "a\nb";
```

"a
b"

A string can span multiple lines, containing literal newlines, but you will get a warning about an "incomplete string" once per newline. Note: the warning won't appear with input at a prompt in the standard interface, only for a code file read in via the read command, or generally in the command-line interface.

```
> "a
  b";
```

"a
b"

New in Maple 2026, there is a new interface setting that allows you to suppress the warning mentioned above:

```
> interface(incompletestringwarning=false):
> "a
  b";
```

"a
b"