Language and Programming

Maple 2021 includes the following enhancements to the Maple language and programming facilities.

Multilevel and conditional next and break statements

Maple's next and break statements provide options to influence loop iteration, ending a single iteration of a loop or a whole loop early. They support new features in Maple 2021.

You can now specify which nested loop it is that you want to break out of early when the condition is met. This is called a *multi-level* break or next statement. It works as follows:

```
> for i to 3 do
    for j to 3 do
      for k to 3 do
        if (i, j, k) = (2, 2, 2) then
          printf("now executing break:\n");
          break j;
        end if;
        printf("%d, %d, %d\n", i, j, k);
      end do;
      printf("end of j-iteration\n");
    end do;
    printf("end of i-iteration\n");
  end do:
```

```
1, 1, 1
1, 1, 2
1, 1, 3
end of j-iteration
1, 2, 1
1, 2, 2
1, 2, 3
end of j-iteration
1, 3, 1
1, 3, 2
1, 3, 3
end of j-iteration
end of i-iteration
2, 1, 1
2, 1, 2
2, 1, 3
end of j-iteration
2, 2, 1
now executing break:
end of i-iteration
3, 1, 1
3, 1, 2
3, 1, 3
end of j-iteration
3, 2, 1
3, 2, 2
3, 2, 3
end of j-iteration
3, 3, 1
3, 3, 2
3, 3, 3
end of j-iteration
end of i-iteration
```

When the **`break j`** statement is executed, Maple immediately ends the loop with **`j`** as the loop variable. Instead of specifying the loop variable, you can also specify a positive integer **N**: this ends the **N**th innermost loop. The previous example could equivalently be specified as **break 2**. Using an integer parameter instead of the loop variable is necessary if the loop you want to terminate does not have a loop variable.

The 'simple' **break** statement without any arguments, the only version that existed before Maple 2021, is equivalent to **break 1**.

The **next** statement also accepts a name or integer to indicate which loop iteration to terminate. We can modify the previous example as follows:

```
> for i to 3 do
    for j to 3 do
      for k to 3 do
        if (i, j, k) = (2, 2, 2) then
          printf("now executing next:\n");
          next j;
        end if;
        printf("%d, %d, %d\n", i, j, k);
      end do;
      printf("end of j-iteration\n");
    end do;
    printf("end of i-iteration\n");
  end do:
```

```
1, 1, 1
1, 1, 2
1, 1, 3
end of j-iteration
1, 2, 1
1, 2, 2
1, 2, 3
end of j-iteration
1, 3, 1
1, 3, 2
1, 3, 3
end of j-iteration
end of i-iteration
2, 1, 1
2, 1, 2
2, 1, 3
end of j-iteration
2, 2, 1
now executing next:
2, 3, 1
```

```
2, 3, 2
2, 3, 3
end of j-iteration
end of i-iteration
3, 1, 1
3, 1, 2
3, 1, 3
end of j-iteration
3, 2, 1
3, 2, 2
3, 2, 3
end of j-iteration
3, 3, 1
3, 3, 2
3, 3, 3
end of j-iteration
end of i-iteration
```

This differs from the previous example in that it continues to the next iteration of the loop over **j**, rather than terminating it entirely.

Another new feature of both statements is that you can append a condition to a **break** or **next** statement, separated by the keyword **if**. This means that the statement is executed only if the condition is true. This feature can be used with simple and multi-level **break** and **next** statements. It is a convenient shorthand that we could have used in the examples above if we omit the printing of "now executing break":

```
> for i to 3 do
    for j to 3 do
      for k to 3 do
        break j if (i, j, k) = (2, 2, 2);
        printf("%d, %d, %d\n", i, j, k);
      end do;
      printf("end of j-iteration\n");
    end do;
    printf("end of i-iteration\n");
  end do:
```

```
1, 1, 1
1, 1, 2
1, 1, 3
end of j-iteration
1, 2, 1
```

```
1, 2, 2
1, 2, 3
end of j-iteration
1, 3, 1
1, 3, 2
1, 3, 3
end of j-iteration
end of i-iteration
2, 1, 1
2, 1, 2
2, 1, 3
end of j-iteration
2, 2, 1
end of i-iteration
3, 1, 1
3, 1, 2
3, 1, 3
end of j-iteration
3, 2, 1
3, 2, 2
3, 2, 3
end of j-iteration
3, 3, 1
3, 3, 2
3, 3, 3
end of j-iteration
end of i-iteration
```

Building Interactive Applications using DocumentTools:-Canvas

The **DocumentTools** package, which provides tools for creating documents and interactive applications programmatically, has been extended in Maple 2021. The new Canvas subpackage provides a framework for Maple users to easily build applications where the number of required input fields is not known in advance. The user can create as many input fields as they need as they use the application, and Maple looks after the details of creating and extracting the information from those fields for you. The new SolvePractice command is a built-in tool that uses the canvas framework in this way. A canvas can also be used to create Maple Learn content in Maple, and then share it through Maple Learn. For details, see Maple Learn Content Tools.

Objects: New calling sequence for methods, and new overloadable built-in functions

When using objects in Maple, you can call methods in various ways: if you want to call a method **m** of an object **o** with extra arguments **a, b, c**, then before Maple 2021, you were able to make this call as **m(o, a, b, c)**, **function:-m(o, a, b, c)**, or **o:-m(o, a, b, c)**. All of those options are of course still available, but Maple 2021 introduces a new, message passing-style, calling mechanism: **o:-m(a, b, c)**. Note the absence of **o** as a parameter. This is only possible if the parameter to be omitted from the calling sequence is named **_self**. For details, see object/methods.

The built-in functions entries and indices, and the new built-in function xormap (see below), now support being overloaded by objects. For details, see object/builtins.

New commands for logic involving sequences

The new xormap command is analogous to the previously existing andmap and ormap commands: it applies a predicate to all operands of an expression and returns the exclusive or of these predicates' results.

```
> xormap(type, [1,2,3,4,5], 'even');
```

$$false \tag{4.1}$$

There are also new commands andseq, orseq, and xorseq, which are similar. Instead of applying a predicate to all operands of an expression, they operate on a sequence of Boolean expressions (at least conceptually; no expression sequence is actually generated). If the values you want to operate on are not already the operands of an expression, then this is typically more convenient and more efficient than using the equivalent **andmap**, **ormap**, and **xormap** commands. (Conversely, if the values you want to operate on *are* already the operands of an expression, then it is typically more convenient and efficient to use the latter commands.)

New options for zip

The zip command applies a function to pairs of values from two containers. It has two new options.

The **evalhf** option instructs Maple to evaluate the function using the evalhf subsystem. It applies only when the two container arguments are rtables of datatype float[8] or complex [8]. This yields a boost in execution speed and a reduction in memory used.

```
> N := 10^6:
  u := Vector(N, i -> i, datatype=float[8]):
  v := Vector(N, i -> N-i, datatype=float[8]):
```

```
> f := (x, y) -> arctan(2*x, y):
```

```
> CodeTools:-Usage(zip(f, u, v));
memory used=227.19MiB, alloc change=144.00MiB, cpu time=2.53s,
real time=2.00s, gc time=1.07s
```

$$\begin{bmatrix} 2.00000199999933 \times 10^{-6} \\ 4.00000799999467 \times 10^{-6} \\ 6.00001799998200 \times 10^{-6} \\ 8.00003199995733 \times 10^{-6} \\ 0.0000100000499999167 \\ 0.0000120000719998560 \\ 0.0000140000979997713 \\ 0.0000160001279996586 \\ 0.0000180001619995140 \\ 0.0000200001999993333 \\ \vdots \end{bmatrix} \tag{5.1}$$

1000000 element Vector[column]

```
> CodeTools:-Usage(zip[evalhf](f, u, v));
memory used=7.63MiB, alloc change=7.63MiB, cpu time=220.00ms,
real time=222.00ms, gc time=0ns
```

$$\begin{bmatrix} 2.00000199999933 \times 10^{-6} \\ 4.00000799999467 \times 10^{-6} \\ 6.00001799998200 \times 10^{-6} \\ 8.00003199995733 \times 10^{-6} \\ 0.0000100000499999167 \\ 0.0000120000719998560 \\ 0.0000140000979997713 \\ 0.0000160001279996586 \\ 0.0000180001619995140 \\ 0.0000200001999993333 \\ \vdots \end{bmatrix} \tag{5.2}$$

1000000 element Vector[column]

The **inplace** option applies only when the two container arguments are rtables of the same size, storage, and data type. When this is used, the first container argument is modified in-place, instead of creating a new container to hold the result. This yields savings in the amount of memory used, especially when combined with the **evalhf** option.

```
> CodeTools:-Usage(zip[evalhf,inplace](f, u, v));
memory used=3.05KiB, alloc change=0 bytes, cpu time=220.00ms,
real time=222.00ms, gc time=0ns
```

$$\left[\begin{array}{c} 2.00000199999933\times10^{-6} \\ 4.00000799999467\times10^{-6} \\ 6.00001799998200\times10^{-6} \\ 8.00003199995733\times10^{-6} \\ 0.0000100000499999167 \\ 0.0000120000719998560 \\ 0.0000140000979997713 \\ 0.0000160001279996586 \\ 0.0000180001619995140 \\ 0.0000200001999993333 \\ \vdots \end{array}\right] \tag{5.3}$$

1000000 element Vector[column]

New options and commands for folding, scanning, and reducing

The commands map, seq, select, remove, and selectremove now each have three new options for combining the resulting expressions using a specified function, called **fold**, **reduce**, and **scan**.
With the **fold** option, you specify a function **F** to apply and an initial value **x**. Conceptually, Maple initializes an internal variable **v := x**. For every (non-NULL) value **y** that the five commands above generate, Maple then runs **v := F(v, y)**. The final value of **v** is returned. Here is an example:

```
> select(type, [22,33,44,55,66,77], 'odd');
```

$$[33, 55, 77] \tag{6.1}$$

```
> select[fold=(F,x)](type, [22,33,44,55,66,77], 'odd');
```

$$F(F(F(x, 33), 55), 77) \tag{6.2}$$

```
> select[fold=(igcd, 0)](type, [22,33,44,55,66,77], 'odd');
```

$$11 \tag{6.3}$$

```
> map[fold=(F, x)](g, [22,33,44,55,66,77]);
```

$$F(F(F(F(F(F(x, g(22)), g(33)), g(44)), g(55)), g(66)), g(77)) \tag{6.4}$$

The **reduce** option is very similar. The only difference is initialization: you do not specify **x**, and **v** is instead initialized to the first value that the command generates (and Maple skips application of **F** for that first value). If no value is generated, Maple returns **undefined**.

```
> select[reduce=F](type, [22,33,44,55,66,77], 'odd');
```

$$F(F(33, 55), 77) \tag{6.5}$$

```
> select[reduce=igcd](type, [22,33,44,55,66,77], 'odd');
```

$$11 \tag{6.6}$$

```
> map[reduce=F](g, [22,33,44,55,66,77]);
```

$$F(F(F(F(F(g(22), g(33)), g(44)), g(55)), g(66)), g(77)) \tag{6.7}$$

The **scan** option is similar to **reduce**, but it returns all intermediate values generated.

```
> select[scan=F](type, [22,33,44,55,66,77], 'odd');
```

$$[33, F(33, 55), F(F(33, 55), 77)] \tag{6.8}$$

```
> select[scan=igcd](type, [22,33,44,55,66,77], 'odd');
```

$$[33, 11, 11] \tag{6.9}$$

```
> map[scan=F](g, [22,33,44]);
```

$$[g(22), F(g(22), g(33)), F(F(g(22), g(33)), g(44))] \tag{6.10}$$

These new options allow for performance improvements that can be significant. Suppose we have a long list of integers and we want to add the even ones.

```
> r := rand(-10^6 .. 10^6):
```

```
> L := [seq(r(), 1 .. 10^6)]:
```

```
> CodeTools:-Usage(add(select(type, L, even)), iterations=10);
memory used=11.45MiB, alloc change=72.54MiB, cpu time=124.00ms,
real time=119.00ms, gc time=9.60ms
```

$$684462136 \tag{6.11}$$

```
> CodeTools:-Usage(select[reduce=`+`](type, L, even), iterations=10)
  ;
```

The second call takes about half the time that the first call takes, but more importantly, it uses only a tiny fraction of the memory.

```
memory used=0.68KiB, alloc change=0 bytes, cpu time=76.00ms, real
time=81.00ms, gc time=0ns
```

$$684462136 \tag{6.12}$$

The new command reduce applies a specified reduction argument to the operands of an expression. Calling `reduce(f, expr)` is equivalent to `seq[reduce=f](i, i = expr)`, but more convenient and efficient.

```
> reduce(F, [33,55,77]);
```

$$F(F(33, 55), 77) \tag{6.13}$$

```
> reduce(igcd, [33,55,77]);
```

$$11 \tag{6.14}$$

The **scan** option has slightly simpler syntax than for the five existing commands, but the result is similar.

```
> reduce[scan](igcd, [33,55,77]);
```

$$[33, 11, 11] \tag{6.15}$$

Finally, the ArrayTools package has two new commands, ReduceAlongDimension and ScanAlongDimension, which do very similar things, but only for rtables and with some options for specialized cases, such as setting the data type of the resulting rtable. They are discussed below, in the section about that package.

Another new option for select, remove, and selectremove

The select, remove, and selectremove commands are discussed above as receiving new options for folding, scanning, and reducing. They have another new option in Maple 2021.

Before Maple 2021, the selecting function submitted to these commands had to accept as its *first* argument the operands from the expression one is selecting from. In Maple 2021, you can use selecting functions that accept these operands as a different argument by using an integer index to the command. For example, if you have a set of types, and you want to know which of these types a given expression has, you can do that as follows:

```
> expr := sqrt(2);
```

$$expr := \sqrt{2} \tag{7.1}$$

```
> types := {numeric, radical, constant, function, complex,
  complexcons};
```

$$types := \{complex, numeric, radical, complexcons, constant, function\} \tag{7.2}$$

```
> applicable, not_applicable := selectremove[2](type, expr, types);
```

$$applicable, not\_applicable := \{complex, radical, complexcons, constant\}, \{numeric, function\} \tag{7.3}$$

Creating evenly spaced sequences

The seq command has a new option, '*numelems*', which allows you to specify the number of elements in a range. It avoids roundoff error, ensures that the endpoint is attained and correctly intersperses the values.

```
> seq( 1..100, numelems=4);
```

$$1, 34, 67, 100 \tag{8.1}$$

```
> seq( 1.1 .. 9.3, numelems = 27 );
```

```
1.1, 1.415384615, 1.730769231, 2.046153846, 2.361538462, 2.676923077, 2.992307692,        (8.2)
    3.307692308, 3.623076923, 3.938461539, 4.253846154, 4.569230769, 4.884615385,
    5.200000000, 5.515384616, 5.830769231, 6.146153846, 6.461538462, 6.776923077,
    7.092307693, 7.407692308, 7.723076923, 8.038461539, 8.353846154, 8.669230770,
    8.984615385, 9.300000000
```

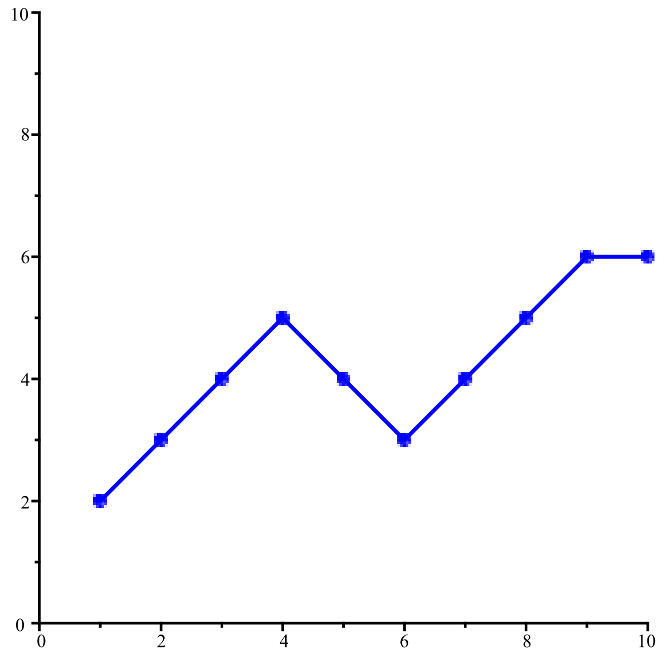
ArrayTools

The new IsMonotonic command in ArrayTools is used to determine if all, or a segment of, values in a 1-D list or container are monotonic (increasing, non-decreasing, decreasing, non-increasing). For example:

```
> with( ArrayTools ):
> A := Array( [ 2, 3, 4, 5, 4, 3, 4, 5, 6, 6 ] );
```

$$A := \left[\begin{array}{cccccccccc} 2 & 3 & 4 & 5 & 4 & 3 & 4 & 5 & 6 & 6 \end{array}\right] \tag{9.1}$$

```
> dataplot( A, view = [ 0 .. 10, 0 .. 10 ], color = blue );
```



```
> IsMonotonic( A, direction = increasing, strict = false );
```

$$\mathit{false} \tag{9.2}$$

```
> IsMonotonic( A, .. 4, direction = increasing, strict = true );
```

$$\mathit{true} \tag{9.3}$$

```
> IsMonotonic( A, 4 .. 6, direction = decreasing, strict = true );
```

$$\mathit{true} \tag{9.4}$$

```
> IsMonotonic( A, 6 .. -1, direction = increasing, strict = true );
```

$$\mathit{false} \tag{9.5}$$

```
> IsMonotonic( A, 6 .. -1, direction = increasing, strict = false )
  ;
```

$$\mathit{true} \tag{9.6}$$

The new command GeneralInnerProduct computes the general inner product of two rtables. This operation could be defined by taking the standard definition of a (real) inner product and replacing multiplication and addition with custom functions. One possible application is in so-called max-plus algebras (also known as tropical matrix algebras). In these algebras over the real numbers together with the symbol $-\infty$, the **max**-operator assumes the role of addition and regular addition assumes the role of multiplication. They are useful, for example, in determining the value of walking the optimal path through a deterministic Markov decision process. Here is an example of a tropical matrix multiplication:

```
> m1, m2 := LinearAlgebra:-RandomMatrix(4), LinearAlgebra:-
  RandomMatrix(4):
  m1[2,4] := -infinity:
  m2[3,3] := -infinity:
  m1, m2;
```

$$\begin{bmatrix} -93 & -32 & 8 & 44 \\ -76 & -74 & 69 & -\infty \\ -72 & -4 & 99 & -31 \\ -2 & 27 & 29 & 67 \end{bmatrix}, \begin{bmatrix} 50 & -50 & -38 & -98 \\ 10 & -22 & -18 & -77 \\ -16 & 45 & -\infty & 57 \\ -9 & -81 & 33 & 27 \end{bmatrix} \tag{9.7}$$

```
> m12 := GeneralInnerProduct(m1, max, `+`, m2);
```

$$m12 := \begin{bmatrix} 35 & 53 & 77 & 71 \\ 53 & 114 & -92 & 126 \\ 83 & 144 & 2 & 156 \\ 58 & 74 & 100 & 94 \end{bmatrix} \tag{9.8}$$

In this result, the top left entry is computed by pairing the entries of the first row of $m1$ with those of the first column of $m2$, taking the sum of each pair. The resulting entry is then the maximum among these four sums.

The trace of a matrix is defined as the sum of its diagonal entries; in the max-plus algebra, this "sum" is computed using the max-operator, for example, as follows:

```
> tm12 := max(LinearAlgebra:-Diagonal(m12));
```

$$tm12 := 114 \tag{9.9}$$

It is well known that the trace of a product of matrices does not depend on the order of multiplication. The proof of this theorem works in any algebra over a commutative ring, such as the max-plus algebra. Consequently, while the individual diagonal elements of *m12* and *m21* are different, their maximal diagonal element is the same:

```
> m21 := GeneralInnerProduct(m2, max, `+`, m1);
  tm21 := max(LinearAlgebra:-Diagonal(m21));
```

$$m21 := \begin{bmatrix} -43 & 18 & 61 & 94 \\ -79 & -22 & 81 & 54 \\ 55 & 84 & 114 & 124 \\ 25 & 54 & 132 & 94 \end{bmatrix}$$

$$tm21 := 114 \tag{9.10}$$

There is a similar GeneralOuterProduct command that computes the general outer product. The commands SuggestedDatatype, SuggestedOrder, and SuggestedSubtype are used in these two commands, but they could also be useful by themselves for users writing similar commands.

Another new **ArrayTools** command is called ScanAlongDimension. It is very similar to the new reduce command with the **scan** option (discussed above), with some options that can be used in specialized cases. It can be used to compute a property of, say, each row of a matrix, returning all intermediate results together with the final result. For example, if you need to access all strings made by concatenating strings from the first few entries of a matrix row, you could use this command, as follows.

```
> m := Matrix(3, 5, (i, j) -> StringTools:-Random(2, 'alpha'));
```

$$m := \begin{bmatrix} \text{"ky"} & \text{"zm"} & \text{"iK"} & \text{"LE"} & \text{"fL"} \\ \text{"mm"} & \text{"fF"} & \text{"Ay"} & \text{"dh"} & \text{"nl"} \\ \text{"fm"} & \text{"MJ"} & \text{"oY"} & \text{"Qs"} & \text{"Fl"} \end{bmatrix} \tag{9.11}$$

```
> ScanAlongDimension(cat, m, 2);
```

$$\begin{bmatrix} \text{"ky"} & \text{"kyzm"} & \text{"kyzmiK"} & \text{"kyzmiKLE"} & \text{"kyzmiKLEfL"} \\ \text{"mm"} & \text{"mmfF"} & \text{"mmfFAy"} & \text{"mmfFAydh"} & \text{"mmfFAydhnl"} \\ \text{"fm"} & \text{"fmMJ"} & \text{"fmMJoY"} & \text{"fmMJoYQs"} & \text{"fmMJoYQsFl"} \end{bmatrix} \tag{9.12}$$

To do the same for each column, you would use the following command.

```
> ScanAlongDimension(cat, m, 1);
```

(9.13)

$$\begin{bmatrix} \text{"ky"} & \text{"zm"} & \text{"iK"} & \text{"LE"} & \text{"fL"} \\ \text{"kymm"} & \text{"zmfF"} & \text{"iKAy"} & \text{"LEdh"} & \text{"fLnl"} \\ \text{"kymmfm"} & \text{"zmfFMJ"} & \text{"iKAyoY"} & \text{"LEdhQs"} & \text{"fLnlFl"} \end{bmatrix} \quad \textbf{(9.13)}$$

There is a similar command, ReduceAlongDimension, similar to the new reduce command without the **scan** option, which will give you the last entry of each row or column in these examples.

```
> ReduceAlongDimension(cat, m, 2);
  ReduceAlongDimension(cat, m, 1);
```

$$\begin{bmatrix} \text{"kyzmiKLEfL"} \\ \text{"mmfFAydhnl"} \\ \text{"fmMJoYQsFl"} \end{bmatrix}$$

$$\begin{bmatrix} \text{"kymmfm"} & \text{"zmfFMJ"} & \text{"iKAyoY"} & \text{"LEdhQs"} & \text{"fLnlFl"} \end{bmatrix} \quad \textbf{(9.14)}$$

These examples can also be done by the reduce command, but if one wants to set, for example, the data type of the resulting rtable, that can only be done using the ArrayTools commands. Consider, for example, the following case:

```
> m := Matrix(5, (i, j) -> trunc(i*5/4+j/2-1), datatype=integer[1])
  ;
```

$$m := \begin{bmatrix} 0 & 1 & 1 & 2 & 2 \\ 2 & 2 & 3 & 3 & 4 \\ 3 & 3 & 4 & 4 & 5 \\ 4 & 5 & 5 & 6 & 6 \\ 5 & 6 & 6 & 7 & 7 \end{bmatrix} \quad \textbf{(9.15)}$$

If we want to create, for each column, the number formed by concatenating its digits, we cannot store the result as an **integer[1]** value. If we try, we get the following error message:

```
> ReduceAlongDimension((i,j) -> 10*i+j, m, 1);
Error, (in ArrayTools:-ReduceAlongDimension) unable to store 
'2345' when datatype=integer[1]
```

We can fix this by specifying a more suitable data type, as follows.

```
> ReduceAlongDimension((i,j) -> 10*i+j, m, 1, 'datatype' = 'integer'
  [2]);
```

$$\begin{bmatrix} 2345 & 12356 & 13456 & 23467 & 24567 \end{bmatrix} \quad \textbf{(9.16)}$$

DEQueue

DEQueue provides a means to construct a double-ended queue. A double-ended queue supports efficiently adding and removing entries from both the front and the back.

```
> dq := DEQueue(4,5,6);
```

$$dq := DEQueue(4, 5, 6) \tag{10.1}$$

```
> push_back(dq, 7);
```

$$DEQueue(4, 5, 6, 7) \tag{10.2}$$

```
> push_front(dq, 3);
```

$$DEQueue(3, 4, 5, 6, 7) \tag{10.3}$$

```
> pop_back(dq);
```

$$7 \tag{10.4}$$

```
> pop_front(dq);
```

$$3 \tag{10.5}$$

These operations also allow use of the DEQueue as an efficient queue or stack.

Many built-in functions, such as **map** and **select**, accept DEQueues. They, and other DEQueue-specific commands, are listed on the DEQueue help page.

ListTools:-Deal and ListTools:-Slice

The new Deal command in the ListTools package is similar to Slice, in that a list (or any 1-D container) is sliced into sublists (or sub-containers) differing in size by at most one, but the slices are formed in a manner analogous to dealing a deck of cards (the list) into hands (the sub-lists). For example:

```
> with( ListTools ):
```

```
> Deck := [ "1C", "9S", "4S", "AS", "QH", "6S", "1D", "2S", "JS",
  "QS", "4H", "9C", "4C", "1S", "6H", "KC", "2C", "4D", "8D", "7C" ]
  ;
```

```
Deck := ["1C", "9S", "4S", "AS", "QH", "6S", "1D", "2S", "JS", "QS", "4H", "9C", "4C", "1S",     (11.1)
    "6H", "KC", "2C", "4D", "8D", "7C"]
```

```
> Deal( Deck, 4 );
```

```
["1C", "QH", "JS", "4C", "2C"], ["9S", "6S", "QS", "1S", "4D"], ["4S", "1D", "4H", "6H",     (11.2)
    "8D"], ["AS", "2S", "9C", "KC", "7C"]
```

The **Deal** command can return specific sublists, too, using the optional **handsreturned** argument:

```
> Deal( Deck, 4, 4 );
```

$$["AS", "2S", "9C", "KC", "7C"] \tag{11.3}$$

```
> Deal( Deck, 4, [ 2, 3 ] );
```

$$["9S", "6S", "QS", "1S", "4D"], ["4S", "1D", "4H", "6H", "8D"] \tag{11.4}$$

Furthermore, the **Slice** command has been updated to support 1-D containers that are not lists, and has a new optional **slicesreturned** argument.

PersistentTable

The new PersistentTable package provides a **connection** object that behaves somewhat like a table, except it is (by default) backed by a file containing an SQLite table. As a consequence, any information stored in the table persists when Maple is shut down or restarted. Furthermore, there is some extra functionality for searching through the stored information.

```
> with(PersistentTable):
```

The **style** option specifies the layout of the table. A persistent table with the following style associates one value with each specified key. For simplicity, we demonstrate here a table stored in memory, without a backing file.

```
> fruits := Open(":memory:", style = [key :: anything, value ::
  anything]);
```

$$fruits := << \text{2-column persistent table at :memory:} >> \tag{12.1}$$

```
> fruits[pear] := peer:
> fruits[apple] := appel:
> fruits[banana] := banaan:
> fruits[orange] := sinaasappel:
> fruits[pear], fruits[banana];
```

$$peer, banaan \tag{12.2}$$

```
> Get(fruits, [apple]);
```

$$appel \tag{12.3}$$

You cannot reference a key that has not been assigned: that leads to the following error.

```
> fruits[grape];
Error, bad index into PersistentTable
> Get(fruits, [grape]);
Error, (in PersistentTable:-Get) bad index into PersistentTable
```

Instead, you can use the **MaybeGet** command to retrieve a value if a corresponding key exists, or a default value if it doesn't.

```
> MaybeGet(fruits, [grape], unknown);
```

$$unknown \qquad (12.4)$$

```
> MaybeGet(fruits, [orange], unknown);
```

$$sinaasappel \qquad (12.5)$$

```
> Close(fruits);
```

Tables with integer, float, or boolean columns support more complicated types of queries. They can also use an index to speed up such queries.

```
> tbl := Open(":memory:", prefix = "example2", style = [n1 ::
  integer, n2 :: integer, a :: float, primarykey = 2, 'index'(n2, a)
  ]);
```

$$tbl := << \text{3-column persistent table at :memory: with prefix example2\_} >> \qquad (12.6)$$

```
> for i to 10 do
    for j to 10 do
      tbl[i, j] := rand(0. .. 1.)();
    end do;
  end do:
```

```
> GetAll(tbl, n2 = 2, a > 0.5);
```

$$\{[2, 2, 0.510832738499999994], [3, 2, 0.509176396800000042], [8, 2, 0.647392884299999971], [9, 2, 0.905484433000000033]\} \qquad (12.7)$$

DataFrame and DataSeries now support entries and indices

The DataFrame and DataSeries objects now support the **entries** and **indices** commands.

```
> restart;
```

```
> ds := DataSeries([4, 5, 6], labels=[A, B, C]);
```

$$ds := \begin{bmatrix} A & 4 \\ B & 5 \\ C & 6 \end{bmatrix} \qquad (13.1)$$

```
> df := DataFrame(Matrix(3, (i, j) -> i-2*j), rows=[A, B, C],
  columns=[a, b, c]);
```

$$df := \left[\begin{array}{cccc} & a & b & c \\ A & -1 & -3 & -5 \\ B & 0 & -2 & -4 \\ C & 1 & -1 & -3 \end{array}\right] \quad (13.2)$$

```
> entries(ds, nolist);
```

$$4, 5, 6 \quad (13.3)$$

```
> entries(ds, pairs);
```

$$A = 4, B = 5, C = 6 \quad (13.4)$$

```
> indices(df);
```

$$[A, a], [B, a], [C, a], [A, b], [B, b], [C, b], [A, c], [B, c], [C, c] \quad (13.5)$$

```
> indices(df, pairs);
```

$$(A, a) = -1, (B, a) = 0, (C, a) = 1, (A, b) = -3, (B, b) = -2, (C, b) = -1, (A, c) = -5, (B, c) = -4, (C, c) = -3 \quad (13.6)$$

Units:-Split

The Units:-Split command takes an expression consisting of an optional coefficient and an optional unit, and returns these two parts separately (or optionally, only one of the two).

```
> c1, u1 := Units:-Split(5*Unit(m/s^2));
```

$$c1, u1 := 5, \frac{\mathrm{m}}{\mathrm{s}^2} \quad (14.1)$$

```
> c2, u2 := Units:-Split(Unit(m/s^2));
```

$$c2, u2 := 1, \frac{\mathrm{m}}{\mathrm{s}^2} \quad (14.2)$$

```
> c3, u3 := Units:-Split(5);
```

$$c3, u3 := 5, 1 \quad (14.3)$$

```
> Units:-Split(5*Unit(m/s^2), output=unit);
```

$$\frac{\mathrm{m}}{\mathrm{s}^2} \quad (14.4)$$

Log function calling sequences

There are two new ways to use log functions with a particular base. It has always been possible to use the calling sequence $\log_{base}(value)$ or $\log[base](value)$ to get the logarithm of $value$ with base $base$. Maple 2021 adds the equivalent calling sequence $\log(value, base)$, and for the case $base = 2$, you can now also use $log2(value)$.

```
> log[3](243), log[2](1024), log[a](a^3) assuming a > 0;
```

$$5, 10, 3 \tag{15.1}$$

```
> log(243, 3), log(1024, 2), log(a^3, a) assuming a > 0;
```

$$5, 10, 3 \tag{15.2}$$

```
> log2(1024);
```

$$10 \tag{15.3}$$

CodeTools extensions

The CodeTools:-RecursiveMembers command returns a list of all members of a module and its submodules.

```
> CodeTools:-RecursiveMembers(CodeTools:-Profiling:-Coverage,
  output=members);
```

$$\{CodeTools:-Profiling:-Coverage:-Percent, CodeTools:-Profiling:-Coverage:-Print, CodeTools:-Profiling:-Coverage, EscapePath, ModuleApply, CodeTools:-Profiling:-Coverage:-PercentSelect, adjustProcLine, functionPercent, percentDefaults, printDefaults, printProcLong, printProcNorm, GroupInputTextfield, CodeTools:-Profiling:-Coverage:-TestCoverageWorksheet, percentSelectDefaults, printProcCompact\} \tag{16.1}$$

The CodeTools:-Profiling:-Coverage:-TestCoverageWorksheet command generates a worksheet that helps ensure good test coverage for Maple code you develop. By default, it creates and opens a separate worksheet, but with the following options, it inserts the worksheet contents into the current worksheet, which makes it easier to demonstrate the command in this worksheet. Everything from the following command until the end of the section is part of the generated worksheet, which is intended to test coverage for a package called **`MyNewPackage`**, by running tests called **`a.tst`** and **`b.tst`**.

The CodeTools:-TestFailures command, also used below, reports the tests that have failed.

```
> CodeTools:-Profiling:-Coverage:-TestCoverageWorksheet
  (MyNewPackage, {"a.tst", "b.tst"}, newsheet=false, insert);
```

```
restart;
```

```
all_entries := CodeTools:-RecursiveMembers(MyNewPackage,
'output'='members'):
```

```
procs := select(type, all_entries, procedure);
```

```
map(CodeTools:-Profiling:-Profile, procs):
```

```
currentdir
```

```
("/home/epostma/work_home/sbox/mnl/help/Maplesoft/updates/Maple202
1/more"):

CodeTools:-Test(1, 0, 'label'="next test file: a.tst", 'quiet',
'boolout'):

read "a.tst";

CodeTools:-Test(1, 0, 'label'="next test file: b.tst", 'quiet',
'boolout'):

read "b.tst";

#map(showstat, procs):

map(CodeTools:-Profiling:-Coverage:-Print, procs):

map2(printf, "%s\n", CodeTools:-TestFailures()):
```

Additional improvements

Maple 2021 includes additional improvements for programming.

The subs command has a member option to limit substitutions to the top-level entries of a container.

The trace command has a new statements option to limit the information shown during execution of a traced procedure to only procedure entry and exit.

evalhf[hfloat] returns a hardware float instead of converting the result to software floating-point, as evalhf typically does.

You can now use the :: operator to assert a type on the control variable of a for loop. See coloncolon.

The exports command now supports options all, type, and method. In particular, **`exports(m, method)`** returns the callable exports of a module m.