`student` — educational tools

Table of contents

Introduction

The `student` library provides a very small collection functions which are supposed to be used in teaching mathematics.

The package functions are called using the package name `student` and the name of the function. E.g., use

```
>> R3 := student::Kn(3,Dom::Real)
```

to create the vector space of the 3-tupel over the field `Dom::Real`. This mechanism avoids naming conflicts with other library functions. If this is found to be inconvenient, then the routines of the `student` package may be exported via `export`. E.g., after calling

```
>> export(student, Kn)
```

the function `student::Kn` can be called directly:

```
>> R3 := Kn(3,Dom::Real)
```

All routines of the `student` package are exported simultaneously by

```
>> export(student)
```

The functions available in the `student` library can be listened with:

```
>> info(student)
```

`student::equateMatrix` – **build a matrix equation**

`student::equateMatrix(A, vars)` returns the matrix equation

$$\begin{pmatrix} a_{11}\cdot x_1 & a_{12}\cdot x_2 & \cdots & a_{1,c-1}\cdot x_r \\ \vdots & \vdots & & \vdots \\ a_{r,1}\cdot x_1 & a_{r2}\cdot x_2 & \cdots & a_{r,c-1}\cdot x_r \end{pmatrix} = \begin{pmatrix} a_{1,c} \\ \vdots \\ a_{r,c} \end{pmatrix}$$

Call(s):

- `student::equateMatrix(A, vars)`

Parameters:

`A` — matrix (of category `Cat::Matrix`) over a `Cat::Field`
`vars` — list of indeterminates

Return Value: an expression of the domain type `DOM_EXPR` and of type `"equal"`.

Related Functions: `linalg::expr2Matrix`, `matrix`

Details:

- `student::equateMatrix`(`A`,`vars`) returns the matrix equation

$$\begin{pmatrix} a_{11}\cdot x_1 & a_{12}\cdot x_2 & \cdots & a_{1,c-1}\cdot x_r \\ \vdots & \vdots & & \vdots \\ a_{r,1}\cdot x_1 & a_{r2}\cdot x_2 & \cdots & a_{r,c-1}\cdot x_r \end{pmatrix} = \begin{pmatrix} a_{1,c} \\ \vdots \\ a_{r,c} \end{pmatrix},$$

  where r and c are the row and column number of `A`, and $x_1, \ldots, x_r$ are the elements of `vars`.

- The number of indeterminates given in `vars` must match the row number of the matrix `A`.

Example 1. Let us construct the equation $A * \overrightarrow{x} = \overrightarrow{b}$. First we construct A and b:

```
>> Ab := matrix( [[1,2,3],[-1,3,0]] )
```

```
+-          -+
|   1, 2, 3  |
|            |
|  -1, 3, 0  |
+-          -+
```

Here we have $A = \begin{pmatrix} 1 & 2 \\ -1 & 3 \end{pmatrix}$ and $b = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$. Now we construct the equation $A * \overrightarrow{x} = \overrightarrow{b}$:

```
>> student::equateMatrix( Ab,[x1,x2] )
```

```
                  +-             -+   +-   -+
                  |   x1 + 2 x2   |   |  3  |
                  |               | = |     |
                  |  - x1 + 3 x2  |   |  0  |
                  +-             -+   +-   -+
```

Example 2. We should be carefull to use the right dimension of the matrix and the indeterminates:

```
>> Ab := matrix( [[1,2,3],[-1,3,0]] )
```

```
                          +-          -+
                          |   1, 2, 3  |
                          |            |
                          |  -1, 3, 0  |
                          +-          -+
>> student::equateMatrix( Ab,[x1,x2,x3] )
```

```
 Error: dimension of matrix and number of vars don't match [stu\
 dent::equateMatrix]
```

Changes:

- In previous MuPAD versions it was also possible to specify the indeterminates in a set. But the elements of a set have no order so in this situation the matrix equation was not uniquely defined.

`student::isFree` – test for linear independence of vectors

`student::isFree`(S) tests if the vectors given in S are linear independend.

Call(s):

- `student::isFree(S)`

Parameters:

S — set or list of vectors (of category `Cat::Matrix`) defined over a `Cat::Field`)

Return Value: either TRUE or FALSE.

Related Functions: `linalg::basis`

Details:

- `student::isFree`(S) gives TRUE if S is free, i.e. the vectors of S are linear independend. Otherwise the value FALSE is returned.

Example 1. We define 3 vectors:

```
>> x := matrix( [[2,3,4]] ):
   y := matrix( [[1,-1,1]] ):
   z := matrix( [[2,3,5]] ):
```

And we ask if x, y and z are linear independend.

```
>> student::isFree( {x,y,z} )
```

TRUE

Hence, the vectors x,y,z are linear independent, and therefore the set x,y,z is a basis of $\mathbb{R}^{\not\vdash}$. Of course the vectors x,y and (x-y) are not linear independent:

```
>> student::isFree( [x,y,x-y] )
```

FALSE

If we have vectors from different vector spaces, `student::isFree` will give an error message:

```
>> zz := matrix( [[2,3,5,6]] ):
   student::isFree( {x,y,zz} )

 Error: set contains incompatible vectors [student::isFree]
```

Changes:

- No changes.

`student::Kn` – the vectorspace of n-tupels over K

The domain `student::Kn` represents the vectorspace of n-tupels over the field F.

Domain:

- `student::Kn(F)`
- `student::Kn(n,F)`

Parameters:

`F` — a field, i.e. a domain of category `Cat::Field`.
`n` — a positive integer

Details:

- The domain `student::Kn` represents the vector space of `n`-tuples over the field `F`. The default value of `n` is `1`. `F` must be a domain of category `Cat::Field`.

Creating Elements:

- `student::Kn(n,F)()`
- `student::Kn(n,F)(listofrows)`
- `student::Kn(n,F)(list)`
- `student::Kn(n,F)(indexfunc)`

Parameters:

`list` — list of vector components.
`listofrows` — list of (at most) `n` rows. Each row is a list of vector components.
`indexfunc` — function or functional expression in two parameters (the row and column index).

Categories:

`Cat::VectorSpace(F)`,`Cat::Matrix(F)`

Related Domains: `Dom::MatrixGroup`

Details:

- Elements of `student::Kn` are contructed by a call to the element constructors of `Dom::MatrixGroup(n,1,F)`. Refer to the corresponding methods of `Dom::MatrixGroup(n,1,F)`.
- The call `student::Kn(n, F)()` returns the n-dimensional zero vector. Note that the zero vector is defined by the entry `"zero"`. See also Example 3.

- `student::Kn(n, F)(listofrows)` creates a vector with `n` components $v_1, v_2, \ldots, v_n$, when `listofrows` is the list $[[v_1], [v_2], \ldots, [v_n]]$. Internally `student::Kn(n, F)(listofrows)` calls `Dom::MatrixGroup(n,1,F)(n,1,list` See there for further information.
- `student::Kn(n,F)(list)` creates the vector with `n` components whose components are the entries of `list`. Internally `student::Kn(n, F)(list)` calls `Dom::MatrixGroup(n,1,F)(n,1,list)`. See there for further information.
- `student::Kn(n,F)(indexfunc)` returns the vector whose i-th component is the value of the function call `indexfunc(i,1)`. Internally `student::Kn(n, F)(indexfunc)` calls `Dom::MatrixGroup(n,1,F)(n,1,indexfunc)`. See there for further information.
- `student::Kn(n, F)` has the domain `Dom::MatrixGroup(n,1,F)` as its super domain, i.e., it inherits each method which is defined by `Dom::MatrixGroup(n,1,F)` and not re-implemented by `student::Kn(n, F)`. Methods described below are re-implemented by `student::Kn`.

Mathematical Methods

Method `_mult`: multiplies with a scalar

`_mult(`*`dom`* `x, `*`any`* `r)`

`_mult(`*`any`* `r ,`*`dom`* `x)`

- If `r` is of type `student::Kn`(n,F) this method returns `FAIL`. Otherwise if there is no method `"scalarMult"(x,r)` for the domain `student::Kn`(n,F) defined, the method `"_mult"` of `Dom::MatrixGroup(n,1,F)` is used to multiply `x` and `r`. In general this means `x` is multiplied with the scalar value `r`.
- By defining the method `scalarMult(x,r)` for the domain `student::Kn` you can overload the `"_mult"` method of `student::Kn`.

Example 1. Let us create the vector space of the 3-tupel over the field `Dom::Real`:

```
>> R3 := student::Kn(3,Dom::Real)
                    student::Kn(3, Dom::Real)
```

Now we create some elements of this domain in different ways:

```
>> u := R3([1,2,3]);
   v := R3([[2],[3],[4]]);
   w := R3()
```

```
+-   -+
|  1  |
|     |
|  2  |
|     |
|  3  |
+-   -+

+-   -+
|  2  |
|     |
|  3  |
|     |
|  4  |
+-   -+

+-   -+
|  0  |
|     |
|  0  |
|     |
|  0  |
+-   -+
```

We perform some calculation with the just created elements. We add the three vectors $\vec{v}$, $\vec{w}$ and $\vec{u}$ of the vectorspace, multiply the vector $\vec{w}$ with the scalar 3 and the vector $\vec{v}$ with the scalar -4:

```
>> v + w + u;
   3*w;
   v*(-4)
```

```
+-   -+
|  3  |
|     |
|  5  |
|     |
|  7  |
+-   -+

+-   -+
|  0  |
|     |
|  0  |
|     |
|  0  |
+-   -+
```

```
+-     -+
|   -8  |
|       |
|  -12  |
|       |
|  -16  |
+-     -+
```

Example 2. Let us see how we can use a function for creating elements of the domain. The function `f` computes the square of the given number. So the entry in the i-th row of the constructed vector will be i^2.

```
>> f := i -> i^2:
   R3 := student::Kn(3,Dom::Real);
   R4 := student::Kn(4,Dom::Real);
   v := R3(f);
   w := R4(f)
```

```
                  student::Kn(3, Dom::Real)

                  student::Kn(4, Dom::Real)

                   +-   -+
                   |  1  |
                   |     |
                   |  4  |
                   |     |
                   |  9  |
                   +-   -+

                  +-    -+
                  |   1  |
                  |      |
                  |   4  |
                  |      |
                  |   9  |
                  |      |
                  |  16  |
                  +-    -+
```

Example 3. The zero vector is defined by the entry `"zero"` as we can see in the following example:

```
>> R3 := student::Kn(3,Dom::Real):
   R3::zero();
   v := R3([[2],[3],[4]]);
   v - R3::zero()
```

$$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$\begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix}$$

$$\begin{pmatrix} 2 \\ 3 \\ 4 \end{pmatrix}$$

Super-Domain: `Dom::MatrixGroup`

Axioms

if `F` has `Ax::canonicalRep`
`Ax::canonicalRep`

Background:

- If the user defines a method `scalarMult` to overload the `"_mult"` method of `student::Kn` he is responsible to define a legal scalar multiplication. This means the defined scalar multiplication has to fulfill that the vector space of `n`-tuples over the field `F` is still a vector space. This is not checked by the domain `student::Kn` themself.

Changes:

- No changes.

`student::plotRiemann` – **plot of a numerical approximation to an integral using rectangles**

`student::plotRiemann(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using rectangles and returns a plot of the numerical process.

Call(s):

- `student::plotRiemann(f, x=a..b<, n><, opt1>, ...)`
- `student::plotRiemann(f, x=a..b<, n>, method<, opt1>, ...)`

Parameters:

`f`	—	functional expression in `x`
`x`	—	identifier
`a, b`	—	arithmetical expressions
`n`	—	a positive integer (number of rectangles)
`method`	—	one of the options *`Left`*, *`Middle`*, or *`Right`*
`opt1`	—	plot option(s) for two-dimensional graphical objects

Options:

`Left`	—	The height of each rectangle is determined by the value of the function at the leftpoint of each interval.
`Middle`	—	The height of each rectangle is determined by the value of the function at the middlepoint of each interval (the default method).
`Right`	—	The height of each rectangle is determined by the value of the function at the rightpoint of each interval.

Return Value: a graphical object of the domain type `plot::Group`.

Related Functions: `plot`, `plot::Group`, `student::plotSimpson`, `student::plotTrapezoid`, `student::riemann`

Details:

- `student::plotRiemann(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using n rectangles and returns a graphical object of the numerical process that can be displayed with the function `plot`.

  The height of each rectangle is determined by the value of the function at the middlepoint of each interval (as with option *`Middle`*).

- With `student::plotRiemann(f, x=a..b, n, Left)`, the height of each rectangle is determined by the value of the function at the left-point of each interval.

  Use option *`Right`*, if the rightpoint of each interval should be taken.

- `n` is the number of rectangles to use. The default value is 4.

- The plot options `opt1, ...` must be valid plot options for two-dimensional graphical objects. See `plot2d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

- The graphical object returned has three operands: a group of the (filled) rectangles, a group of rectangles for the frames of the filled rectangles, as well as the function graph of `f` (of the domain type `plot::Function2d`). The first two operands are objects of the domain `plot::Group`.

Example 1. The following call returns a visualization of the numerical approximation to the integral $\int_{-1}^{1} e^x \, dx$ using 10 rectangles:

```
>> p := student::plotRiemann(exp(x), x = -1..1, 10)

                              plot::Group()

>> plot(p)
```

Example 2. You can change plot parameters of the visualization returned by `student::plotRiemann`. For example, to change the color of the filled rectangles to blue, we must set the plot option `Color` of the first operand of `p` to the value `RGB::Blue`:

```
>> (p[1])::Color := RGB::Blue:
   plot(p, Axes = Box)
```

Here we changed the style of the axes of the graphical scene to the value `Box`.

Changes:

- `student::plotRiemann` is a new function.

`student::plotSimpson` – plot of a numerical approximation to an integral using Simpson's rule

`student::plotSimpson(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using Simpson's rule and returns a plot of the numerical process.

Call(s):

- `student::plotSimpson(f, x=a..b<, n><, opt1>, ...)`

Parameters:

`f`	— functional expression in `x`
`x`	— identifier
`a, b`	— arithmetical expressions
`n`	— a positive integer (number of stripes to use)
`opt1`	— plot option(s) for two-dimensional graphical objects

Return Value: a graphical object of the domain type `plot::Group`.

Related Functions: `plot`, `plot::Group`, `student::plotRiemann`, `student::plotTrapezoid`, `student::simpson`

Details:

- `student::plotSimpson(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using Simpson's rule and returns a graphical object of the numerical process that can be displayed with the function `plot`.
- `n` is the number of stripes to use. The default value is 4.
- The plot options `opt1, ...` must be valid plot options for two-dimensional graphical objects. See `plot2d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

- The graphical object returned has `n+1` operands: the n stripes as well as the function graph of `f` (of the domain type `plot::Function2d`). Every stripe is an object of the domain type `plot::Group`.

Example 1. The following call returns a visualization of the numerical approximation to the integral $\int_0^1 \sin(x)\,dx$ using Simpson's rule and 10 stripes:

```
>> p := student::plotSimpson(sin(x), x = 0..1, 10)
```

```
plot::Group()
```

To display it on the screen, call:

```
>> plot(p)
```

Example 2. You can change plot parameters of the visualization returned by `student::plotSimpson`. For example, to change the color of every second filled stripe to red, we must set the plot option `Color` of the operands of `p` with even index to the value `RGB::Blue`:

```
>> ((p[2*i])::Color := RGB::Red) $ i = 1..nops(p) div 2:
   plot(p)
```

Changes:

- `student::plotSimpson` is a new function.

`student::plotTrapezoid` – plot of a numerical approximation to an integral using the Trapezoidal rule

`student::plotTrapezoid(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using the Trapezoidal rule and returns a plot of the numerical process.

Call(s):

- `student::plotTrapezoid(f, x=a..b<, n><, opt1>, ...)`

Parameters:

`f`	—	functional expression in `x`
`x`	—	identifier
`a, b`	—	arithmetical expressions
`n`	—	a positive integer (number of trapezoids to use)
`opt1`	—	plot option(s) for two-dimensional graphical objects

Return Value: a graphical object of the domain type `plot::Group`.

Related Functions: `plot`, `plot::Group`, `student::plotRiemann`, `student::plotTrapezoid`, `student::trapezoid`

Details:

- `student::plotTrapezoid(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using the Trapezoidal rule and returns a graphical object of the numerical process that can be displayed with the function `plot`.
- `n` is the number of trapezoids to use. The default value is 4.
- The plot options `opt1, ...` must be valid plot options for two-dimensional graphical objects. See `plot2d` for details.

  Note that scene options are not allowed! You may give scene options as optional arguments for the function `plot`, or use `plot::Scene` to create an object representing a graphical scene.

- The graphical object returned has three operands: a group of the (filled) trapezoids, a group of polygons representing the frames of the trapezoids, as well as the function graph of `f` (of the domain type `plot::Function2d`). The first two operands are objects of the domain `plot::Group`.

Example 1. The following call returns a visualization of the numerical approximation to the integral $\int_0^{\frac{\pi}{2}} \cos(x)\,dx = 1$ using the Trapezoidal rule and 10 trapezoids:

```
>> p := student::plotTrapezoid(cos(x), x = 0..PI/2, 10)

                              plot::Group()
```

To display it on the screen, call:

```
>> plot(p)
```

Example 2. You can change plot parameters of the visualization returned by `student::plotTrapezoid`. For example, to change the x-range of the graph of f, we set the attribute `range` of the last operand of `p` to the value `x = -PI/2..PI/2`:

```
>> (p[nops(p)])::range := x = -PI/2..PI/2:
   plot(p)
```

Changes:

- `student::plotTrapezoid` is a new function.

`student::riemann` – numerical approximation to an integral using rectangles

`student::riemann(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using rectangles.

Call(s):

- `student::riemann(f, x=a..b<, n>)`
- `student::riemann(f, x=a..b<, n>, method)`

Parameters:

`f`	—	arithmetical expression or a function in `x`
`x`	—	identifier
`a, b`	—	arithmetical expressions
`n`	—	a positive integer (number of rectangles)
`method`	—	one of the options *`Left`*, *`Middle`*, or *`Right`*

Options:

`Left`	—	The height of each rectangle is determined by the value of the function at the leftpoint of each interval.
`Middle`	—	The height of each rectangle is determined by the value of the function at the middlepoint of each interval (the default method).
`Right`	—	The height of each rectangle is determined by the value of the function at the rightpoint of each interval.

Return Value: an arithmetical expression.

Related Functions: `freeze`, `int`, `numeric::int`, `numeric::quadrature`, `student::plotRiemann`, `student::simpson`, `student::trapezoid`

Details:

- `student::riemann(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using n rectangles.

  The height of each rectangle is determined by the value of the function at the middlepoint of each interval (as with option *`Middle`*).

- With `student::riemann(f, x=a..b, n, Left)`, the height of each rectangle is determined by the value of the function at the leftpoint of each interval.

  Use option *`Right`*, if the rightpoint of each interval should be taken.

- `n` is the number of rectangles to use. The default value is 4.

- The result of `student::riemann` is an arithmetical expression which consists of frozen subexpressions of type `"sum"`.

  Use `unfreeze` to force the evaluation of the result.

Example 1. The numerical approximation to the integral $\int_{-1}^{1} e^x \, dx$ using 10 rectangles is:

```
>> student::riemann(exp(x), x = -1..1, 10)
```

$$\mathrm{sum}\left(\exp\left(\frac{i1}{5} - \frac{9}{10}\right), i1 = 0..9\right) \Big/ 5$$

The function values were taken at the middlepoint of each interval, the same as with option *`Middle`*.

We got an unevaluated expression, the formula for the corresponding approximation. Use `unfreeze` to force the evaluation of the result:

```
>> unfreeze(%)
```

$$\frac{\exp(-1/2) + \exp(1/2) + \exp(-1/10) + \exp(1/10) + \exp(-3/10)}{5} + \frac{\exp(3/10) + \exp(-7/10) + \exp(7/10) + \exp(-9/10) + \exp(9/10)}{5}$$

Let us compute a floating-point approximation of the result:

```
>> float(%)
```

$$2.346489615$$

and compare the result with the approximation using the left- and rightpoint of each interval for the determination of the heights of the rectangles:

```
>> float(student::riemann(exp(x), x = -1..1, 10, Left)),
   float(student::riemann(exp(x), x = -1..1, 10, Right));

                  2.123191605, 2.593272083
```

Finally, we compute the exact value of the definite integral $\int_{-1}^{1} e^x \, dx$:

```
>> F:= int(exp(x), x = -1..1); float(F)

                        exp(1) - exp(-1)

                           2.350402387
```

Example 2. The general formula of an approximation of $\int_a^b f(x) \, dx$ using 4 rectangles:

```
>> F:= student::riemann(f(x), x = a..b)

    / b   a \    /  /                / b   a \ \            \
    | - - - | sum| f| a + (i4 + 1/2) | - - - | |, i4 = 0..3 |
    \ 4   4 /    \  \                \ 4   4 / /            /
```

To expand the frozen sum, enter:

```
>> F:= unfreeze(F)

 / b   a \ /  / a   7 b \    / 3 a   5 b \    / 5 a   3 b \
 | - - - | | f| - + --- | + f| --- + --- | + f| --- + --- | +
 \ 4   4 / \  \ 8    8  /    \  8     8  /    \  8     8  /

     / 7 a   b \ \
    f| --- + - | |
     \  8    8 / /
```

Changes:

- `student::riemann` is a new function.

`student::simpson` – numerical approximation to an integral using Simpson's rule

`student::simpson(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x) \, dx$ using Simpson's rule.

Call(s):

- `student::simpson(f, x=a..b<, n>)`

Parameters:

`f`	—	arithmetical expression or a function in `x`
`x`	—	identifier
`a, b`	—	arithmetical expressions
`n`	—	a positive integer (number of stripes to use)

Return Value: an arithmetical expression.

Related Functions: `freeze`, `int`, `numeric::int`, `numeric::quadrature`, `student::plotSimpson`, `student::riemann`, `student::trapezoid`

Details:

- `student::simpson(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using Simpson's rule.
- `n` is the number of stripes to use. The default value is 4.
- The result of `student::simpson` is an arithmetical expression which consists of frozen subexpressions of type `"sum"`.

  Use `unfreeze` to force the evaluation of the result.

Example 1. The numerical approximation to the integral $\int_0^1 \sin(x)\,dx$ using Simpson's rule and 10 stripes is:

```
>> student::simpson(sin(x), x = 0..1, 10)
```

$$\frac{\sin(1)}{30} + \frac{\displaystyle\sum_{i1=1..4}\left(\sin\left(\frac{i1}{5}\right)\right)}{15} + \frac{2\displaystyle\sum_{i1=1..5}\left(\sin\left(\frac{i1}{5} - \frac{1}{10}\right)\right)}{15}$$

We got an unevaluated expression, the formula for the corresponding approximation. Use `unfreeze` to force the evaluation of the result:

```
>> unfreeze(%)
```

```
 sin(1)   2 sin(1/2)   sin(1/5)   sin(2/5)   sin(3/5)
 ------ + ---------- + -------- + -------- + -------- +
   30         15          15         15         15

    sin(4/5)   2 sin(1/10)   2 sin(3/10)   2 sin(7/10)
    -------- + ----------- + ----------- + ----------- +
       15          15            15            15

    2 sin(9/10)
    -----------
        15
```

Let us compute a floating-point approximation of the result:

```
>> float(%)
```

```
                          0.4596979498
```

and compare it with the exact value of the definite integral $\int_0^1 \sin(x)\,dx$:

```
>> F:= int(sin(x), x = 0..1); float(F)
```

```
                           1 - cos(1)

                          0.4596976941
```

Example 2. The general formula of Simpson's rule (using 4 stripes):

```
>> F:= student::simpson(f(x), x = a..b)
```

```
 / b    a  \ /                  / a   b \
 | -- - -- | | f(a) + f(b) + 2 f| - + - | +
 \ 12   12 / \                  \ 2   2 /

         /  /                / b   a \ \           \ \
    4 sum| f| a + (2 i3 - 1) | - - - | |, i3 = 1..2 | |
         \  \                \ 4   4 / /           / /
```

To expand the frozen sum, enter:

```
>> F:= unfreeze(F)
```

```
 / b    a  \ /                  / a   b \      / a   3 b \
 | -- - -- | | f(a) + f(b) + 2 f| - + - | + 4 f| - + --- | +
 \ 12   12 / \                  \ 2   2 /      \ 4    4  /

       / 3 a   b \ \
    4 f| --- + - | |
       \  4    4 / /
```

You may even expand this product:

```
>> expand( F )
```

$$\frac{b\,f(a)}{12} - \frac{a\,f(b)}{12} - \frac{a\,f(a)}{12} + \frac{b\,f(b)}{12} - \frac{a\,f\left|\frac{a}{2} - \frac{b}{2}\right.}{6}$$

$$\frac{b\,f\left|\frac{a}{2} - \frac{b}{2}\right.}{6} - \frac{a\,f\left|\frac{a}{4} - \frac{3\,b}{4}\right.}{3} - \frac{a\,f\left|\frac{3\,a}{4} - \frac{b}{4}\right.}{3}$$

$$\frac{b\,f\left|\frac{a}{4} - \frac{3\,b}{4}\right.}{3} + \frac{b\,f\left|\frac{3\,a}{4} - \frac{b}{4}\right.}{3}$$

Changes:

- `student::simpson` is a new function.

`student::trapezoid` – numerical approximation to an integral using the Trapezoidal rule

`student::trapezoid(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using the Trapezoidal rule.

Call(s):

- `student::trapezoid(f, x=a..b<, n>)`

Parameters:

`f`	— arithmetical expression or a function in `x`
`x`	— identifier
`a, b`	— arithmetical expressions
`n`	— a positive integer (number of trapezoids to use)

Return Value: an arithmetical expression.

Related Functions: `freeze`, `int`, `numeric::int`, `numeric::quadrature`, `student::plotTrapezoid`, `student::riemann`, `student::simpson`

Details:

- `student::trapezoid(f, x=a..b, n)` computes a numerical approximation to the integral $\int_a^b f(x)\,dx$ using the Trapezoidal rule.
- `n` is the number of trapezoids to use. The default value is 4.
- The result of `student::trapezoid` is an arithmetical expression which consists of frozen subexpressions of type `"sum"`.

  Use `unfreeze` to force the evaluation of the result.

Example 1. The numerical approximation to the integral $\int_0^{\frac{\pi}{2}} \cos(x)\,dx = 1$ using the Trapezoidal rule and 10 trapezoids is:

```
>> student::trapezoid(cos(x), x = 0..PI/2, 10)
```

$$\frac{PI}{40}\left(2\sum_{i1=1..9}\cos\left(\frac{PI\ i1}{20}\right)+1\right)$$

We got an unevaluated expression, the formula for the corresponding approximation. Use `unfreeze` to force the evaluation of the result:

```
>> unfreeze(%)
```

$$\frac{PI}{40}\left(2\cos\left(\frac{PI}{20}\right)+2\cos\left(\frac{3\,PI}{20}\right)+2\cos\left(\frac{7\,PI}{20}\right)+2\cos\left(\frac{9\,PI}{20}\right)+2+5+\frac{2\,(5+5^{1/2})^{1/2}}{2}+\frac{2\,(5-5^{1/2})^{1/2}}{2}+1\right)$$

Let us compute a floating-point approximation of the result:

```
>> float(%)
```

```
                                0.9979429864
```

Example 2. The general formula of the Trapezoidal rule (using 4 trapezoids):

```
>> F:= student::trapezoid(f(x), x = a..b)
```

```
 / b   a \ /                    /  /        / b   a \ \
 | - - - | | f(a) + f(b) + 2 sum| f| a + i2 | - - - | |,
 \ 8   8 / \                    \  \        \ 4   4 / /

               \ \
    i2 = 1..3  | |
               / /
```

To expand the frozen sum, enter:

```
>> F:= unfreeze(F)
```

```
 / b   a \ /                   / a   b \      / a   3 b \
 | - - - | | f(a) + f(b) + 2 f| - + - | + 2 f| - + --- | +
 \ 8   8 / \                   \ 2   2 /      \ 4    4  /

       / 3 a   b \ \
    2 f| --- + - | |
       \  4    4 / /
```

Changes:

- `student::trapezoid` is a new function.