`stats` — library for statistics

Table of contents

1 Statistics

The `stats` package provides methods for statistical analysis.

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

```
>> stats::mean(..data..)
```

to compute the mean of statistical data. This mechanism avoids naming conflicts with other library functions. If this is found to be inconvenient, then the routines of the `stats` package may be exported via `export`. E.g., after calling

```
>> export(stats, mean)
```

the function `stats::mean` may be called directly:

```
>> mean(..data..)
```

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

```
>> export(stats)
```

Note, however, that a naming conflict with the function `normal` of the standard library exists. The normal distribution, implemented by `stats::normal`, is therefore not exported by the above call.

Further, if the identifier `mean`, say, already has a value, then `export` returns a warning and does not export `stats::mean`. The value of the identifier `mean` must be deleted before it can be exported successfully from the `stats` package.

`stats::BPCorr` – **Bravais-Pearson correlation**

`stats::BPCorr(data)` returns the Bravais-Pearson correlation coefficient of data pairs.

Call(s):

- `stats::BPCorr([x1, x2, ..], [y1, y2, ..])`
- `stats::BPCorr([[x1, y1], [x2, y2], ..])`
- `stats::BPCorr(s <, cx, cy>)`
- `stats::BPCorr(s <, [cx, cy]>)`

Parameters:

`x1, x2, ..`	—	statistical data: arithmetical expressions.
`y1, y2, ..`	—	statistical data: arithmetical expressions.
`s`	—	a sample of domain type `stats::sample`.
`cx, cy`	—	integers representing column indices of the sample `s`. Column `cx` provides the data `x1, x2, ..`, column `cy` provides the data `y1, y2, ...`

Return Value: an arithmetical expression. `FAIL` is returned, if the Bravais-Pearson correlation coefficient does not exist.

Related Functions: `stats::FCorr`, `stats::sample`

Details:

- The Bravais-Pearson correlation coefficient of data pairs (x_i, y_i) is given by

$$\frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\left(\sum_i (x_i - \bar{x})^2\right)\ \left(\sum_i (y_i - \bar{y})^2\right)}},$$

  where $\bar{x}$ and $\bar{y}$ are the mean values of the data x_i and y_i, respectively.

  This coefficient is a number between -1 and 1. It is close to 1, if the relation is approximately positive linear. It is close to -1, if it is negative linear. Values close to 0 correspond to non-linear relations or to unrelated data, respectively.

- The column indices `cx`, `cy` are optional, if the data are given by a sample object containing only two non-string columns. Cf. example 2.

Example 1. We calculate the Bravais-Pearson correlation coefficient of four pairs of values given in two lists. There is a positive linear relation $y = 1 + 2x$ between the entries of the lists:

```
>> stats::BPCorr([0, 1, 2, 3], [1, 3, 5, 7])

                                  1
```

Alternatively, the data may be specified by a list of pairs:

```
>> stats::BPCorr([[0, 0], [1, -3], [2, -4], [3, -3]])

                                 1/2
                                5
                              - ----
                                 3

>> float(%)

                           -0.7453559925
```

Example 2. We create a sample consisting of one string column and two non-string columns:

```
>> stats::sample([["a", 0, 0], ["b", 10, 10], ["c", 20, 35]])

                           "a"   0   0
                           "b"  10  10
                           "c"  20  35
```

The Bravais-Pearson correlation coefficient is calculated using the data columns 2 and 3. In this example there are only two non-string columns, so the column indices do not have to be specified:

```
>> float(stats::BPCorr(%))

                           0.9707253434
```

Example 3. We create a sample consisting of three data columns:

```
>> stats::sample([[1, 0, 0], [2, 10, 10], [3, 20, 35]])

                            1   0   0
                            2  10  10
                            3  20  35
```

We compute the Bravais-Pearson correlation coefficient of the data pairs given by the first and the second column:

```
>> stats::BPCorr(%, 1, 2)
```

$$1$$

This result indicates that there is a linear relation between these columns. Indeed, the i-th entry y of column 2 is given by $y = 10\,(x - 1)$, where x is the i-th entry of column 1.

Example 4. We create a sample of three columns containing symbolic data:

```
>> stats::sample([[1, a, 10], [2, 10, A], [3, 6, 30], [x, 30, 10]])
```

1	a	10
2	10	A
3	6	30
x	30	10

We compute the Bravais-Pearson correlation coefficient of the data pairs given by the second and the third column. Here we specify these columns by a list of column indices:

```
>> stats::BPCorr(%, [2, 3])
```

$$\left/ \quad \left/ A \right. \cdot a - 4 \right| - + \frac{25}{2} \left| \left| - + \frac{23}{2} \right| + 480 \right| /$$
$$\backslash \quad \backslash \quad 4 \quad / \backslash \quad 4 \quad / \quad /$$

$$\left/ / \quad 2 \right. \left| a - 4 \right| - + \frac{23}{2} \left| + 1036 \right| /$$
$$\backslash \backslash \quad \backslash \quad 4 \quad / \quad /$$

$$\left/ \quad 2 \right. \left| A - 4 \right| - + \frac{25}{2} \left| + 1100 \right| \left| \right.^{1/2}$$
$$\backslash \quad \backslash \quad 4 \quad / \quad / \ /$$

Changes:

- `stats::BPCorr` is a new function.

`stats::FCorr` – Fechner correlation

`stats::FCorr(data)` returns the Fechner correlation coefficient of data pairs.

Call(s):

- `stats::FCorr([x1, x2, ..], [y1, y2, ..])`
- `stats::FCorr([[x1, y1], [x2, y2], ..])`
- `stats::FCorr(s <, cx, cy>)`
- `stats::FCorr(s <, [cx, cy]>)`

Parameters:

`x1, x2, ..`	— statistical data: arithmetical expressions.
`y1, y2, ..`	— statistical data: arithmetical expressions.
`s`	— a sample of domain type `stats::sample`.
`cx, cy`	— integers representing column indices of the sample `s`. Column `cx` provides the data `x1, x2, ..`, column `cy` provides the data `y1, y2, ...`

Return Value: an arithmetical expression. `FAIL` is returned, if the data are empty.

Related Functions: `stats::BPCorr`, `stats::sample`

Details:

- The Fechner correlation coefficient of n data pairs (x_i, y_i) is given by $(2 \cdot V - n)/n$ with $V = \sum_i v_i$. The number v_i is 1, if $x_i - \bar{x}$ and $y_i - \bar{y}$ have the same sign or are both 0. It is $1/2$, if either $x_i - \bar{x}$ or $y_i - \bar{y}$ is 0. It is 0 in all other cases. Here $\bar{x}$ and $\bar{y}$ are the mean values of the data x_i and y_i, respectively.

  The Fechner correlation coefficient is a number between -1 and 1. It is positive for a positive linear relation and negative for a negative linear relation.

- The column indices `cx`, `cy` are optional, if the data are given by a sample object containing only two non-string columns. Cf. example 2.

- The Fechner correlation should not be computed for symbolic data. These may lead to unexpected results, if the sign of symbolic parameters cannot be determined.

Example 1. We calculate the Fechner correlation coefficient of four data pairs given in two lists. There is a positive linear relation $y = 1 + 2x$ between the entries of the lists:

```
>> stats::FCorr([0, 1, 2, 3], [1, 3, 5, 7])
```

```
                                  1
```

Alternatively, the data may be specified by a list of pairs:

```
>> stats::FCorr([[0, 0], [1, -3], [2, -4], [3, -3]])
```

```
-1/2
```

Example 2. We create a sample consisting of one string column and two non-string columns:

```
>> stats::sample([["a", 0, 0], ["b", 10, 10], ["c", 20, 35]])
```

```
"a"   0   0
"b"  10  10
"c"  20  35
```

The Fechner correlation coefficient is calculated using the data columns 2 and 3. In this example there are only two non-string columns, so the column indices do not have to be specified:

```
>> stats::FCorr(%)
```

```
2/3
```

Example 3. We create a sample consisting of three data columns:

```
>> stats::sample([[1, 0, 0], [2, 10, 10], [3, 20, 35]])
```

```
1   0   0
2  10  10
3  20  35
```

We compute the Fechner correlation coefficient of the data pairs given by the first and the second column:

```
>> stats::FCorr(%, 1, 2)
```

```
1
```

Example 4. We create a sample consisting of three columns:

```
>> stats::sample([[1, -3, 1], [2, -8, 3], [3, -12, 5], [5, 10, 7]])
```

```
1   -3  1
2   -8  3
3  -12  5
5   10  7
```

We compute the Fechner correlation coefficient of the data pairs given by the second and the third column. Here we specify these columns by a list of column indices:

```
>> stats::FCorr(%, [1, 2])
```

```
0
```

Changes:

- `stats::FCorr` is a new function.

`stats::Tdist` – the T-distribution

`stats::Tdist(x, v)` computes the value

$$\frac{\Gamma(\frac{v+1}{2})}{\sqrt{\pi\, v}\,\Gamma(\frac{v}{2})} \int_{-\infty}^{x} \frac{dt}{(1+\frac{t^2}{v})^{(v+1)/2}}$$

of the T-distribution with v degrees of freedom at the point x.

Call(s):

- `stats::Tdist(x, v)`

Parameters:

`x` — an arithmetical expression.
`v` — the "degrees of freedom": a positive real value.

Return Value: If x is a real float and v is a constant real value, then a floating point value is returned. If x is not a real float and v is a positive integer, then an explicit arithmetical expression is returned. In all other cases an unevaluated call of `stats::Tdist` is returned.

Side Effects: The function is sensitive to the environment variable `DIGITS`, when the argument `x` is a floating point number.

Related Functions: `gamma`, `stats::normal`, `stats::ChiSquare`

Details:

- The integral is evaluated by a recursive formula, if v is a positive integer and x is not a real float. If floating point approximations are desired for integer v and exact numerical values of x such as `PI/2 + sqrt(2)`, `exp(-2)` etc., then we strongly recommend to use `stats::Tdist(float(x), v)` rather than `float(stats::Tdist(x, v))`. This avoids the overhead of the intermediate symbolic result `stats::Tdist(x, v)`. Cf. example 3.

Example 1. We compute the T-distribution with two degrees of freedom at the point $x = 0.5$:

```
>> stats::Tdist(0.5, 2)
```

```
                          0.6666666667
```

Exact values are produced, if the first argument is exact and the second argument is a positive integer:

```
>> stats::Tdist(7/8, 2)
```

```
                     /             1/2    1/2 \
                 1/2 |  1/2   7 128    177    |
                2    | 2    + --------------- |
                     \             1416       /
                    -------------------------------
                                  4
```

Example 2. We compute the T-distribution with three degrees of freedom with a symbolic argument. An explicit expression is returned, when the second argument is a positive integer:

```
>> stats::Tdist(x, 3)
```

```
          /                                      /    1/2 \ \
          |                            1/2       | x 3    | |
          |     1/2                   3    arctan| ------ | |
      1/2 | PI 3           x                     \   3    / |
    2 3   | ------- + ------------ + --------------------
- |
          |    4        /  2     \             2           |
          |             | x      |                         |
          |           2 | -- + 1 |                         |
          \             \ 3      /                        /
    ------------------------------------------------------
---
                              3 PI
```

An unevaluated call is returned, if the second argument is not a positive integer:

```
>> stats::Tdist(x, 3/2)
```

```
                     stats::Tdist(x, 3/2)
```

```
>> diff(%, x)
```

```
                            1/2  1/2
                           PI    3
                  ------------------------------
                                /    2     \5/4
                               2 | 2 x      |
                  6 gamma(3/4)  | ---- + 1 |
                                \  3       /
```

Example 3. We compute a floating point value via an intermediate exact result:

```
>> stats::Tdist(7/2, 20)
```

```
 /            /         1/2
 |        1/2 | 131072 5
 | 46189 5    | ----------- +
 \            \   230945

                                  1/2    1/2 \ \
    367342933035385754953318480    129    | |
    ------------------------------------- | | / 262144
         29471732531300365234637494 5     / /
```

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

```
                          0.9988724384
```

It is more efficient to compute this value by a direct call with a floating point argument:

```
>> stats::Tdist(float(7/2), 20)
```

```
                          0.9988724384
```

Changes:

- Floating point evaluation was made more efficient. Further, it was extended to arbitrary real positive "degrees of freedom".

`stats::a_quantil` – α-quantile of discrete data

`stats::a_quantil(a, ..)` returns the α-quantile of discrete data.

Call(s):

- `stats::a_quantil(a, x1, x2, ..)`
- `stats::a_quantil(a, [x1, x2, ..])`
- `stats::a_quantil(a, s <, c>)`

Parameters:

`a`	—	the α value: a rational number or a real float from the interval $(0, 1)$.
`x1, x2, ..`	—	the statistical data: real numerical values.
`s`	—	a sample of domain type `stats::sample`.
`c`	—	an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Value: an arithmetical expression. `FAIL` is returned, if the data are empty.

Related Functions: `stats::geometric`, `stats::harmonic`, `stats::mean`, `stats::median`, `stats::modal`, `stats::quadratic`, `stats::stdev`, `stats::variance`

Details:

- The α-quantile of n sorted values $x_1 \leq \cdots \leq x_n$ is $(x_k + x_{k+1})/2$, if $k := n\,\alpha$ is an integer. If k is not an integer, then the α-quantile is the data element $x_{\text{ceil}(k)}$, where `ceil`(k) is the next integer larger than k.
- The data do not have to be sorted on input: `stats::a_quantil` sorts internally.
- The data must be real and numerical. Expressions such as `PI + sqrt(2)`, `exp(-5)` etc. are converted to floating point numbers.
- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.
- The $\frac{1}{2}$-quantile is called median. This special quantile is implemented in `stats::median`.

Example 1. We calculate the $\frac{1}{4}$-quantile of a sequence of five values:

```
>> stats::a_quantil(1/4, 3, 8, 5, 9/2, 11)
```

```
                                9/2
```

Alternatively, the data may be passed as a list:

```
>> stats::a_quantil(1/4, [3, 8, 5, 9/2, 11])
```

```
                                9/2
```

Example 2. We create a sample:

```
>> stats::sample([[4, 7, 5], [3, 6, 17], [8, 2, 2]])
```

```
                            4  7   5
                            3  6  17
                            8  2   2
```

The $\frac{1}{2}$-quantile (the `median`) of the second column is calculated:

```
>> stats::a_quantil(1/2, %, 2)
```

```
                                6
```

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
                          "1996"  1242
                          "1997"  1353
                          "1998"  1142
```

The 0.3-quantile of the second column is calculated. In this case this column does not have to be specified, since it is the only non-string column in the sample:

```
>> stats::a_quantil(0.3, %)
```

```
                               1142
```

Changes:

- `stats::a_quantil` is a new function.

`stats::calc` – apply functions to samples

`stats::calc(s, ..)` applies functions to columns of the sample `s`.

Call(s):

- `stats::calc(s, c, f1 <, f2, ..>)`
- `stats::calc(s, [c1, c2, ..], f1 <, f2, ..>)`

Parameters:

`s`	— a sample of domain type `stats::sample`.
`c, c1, c2, ..`	— positive integers representing column indices of the sample.
`f1, f2, ..`	— procedures.

Return Value: a sample of domain type `stats::sample`.

Related Functions: `stats::tabulate`

Details:

- In a call such as `stats::calc(s, c, f1)` the function `f1` is applied to the elements of the column `c` of `s`. This generates a new column which is appended to `s`. If present, the next function `f2` is applied to the new sample etc. Thus, a call of `stats::calc` with m functions appends m new columns to `s`.

  Each function must accept one parameter.

- In a call such as `stats::calc(s, [c1, c2, ..], f1)` the i-th element of the new column is given by `f1(s[i, c1]), s[i, c2], ..)`.

  Each function must accept as many parameters as specified by the second argument of `stats::calc`.

Example 1. We create a sample of three rows and three columns:

```
>> stats::sample([[1, a1, b1], [2, a2, b2], [3, a3, b3]])
```

```
1  a1  b1
2  a2  b2
3  a3  b3
```

We add and multiply the elements of the columns 2 and 3 by applying the system functions `_plus` and `_mult`:

```
>> stats::calc(%, [2, 3], _plus, _mult)
```

```
                  1  a1  b1  a1 + b1  a1*b1
                  2  a2  b2  a2 + b2  a2*b2
                  3  a3  b3  a3 + b3  a3*b3
```

The following call maps each element of the second column of the original sample to its fourth power:

```
>> stats::calc(%2, 2, x -> x^4)
```

```
                        1  a1  b1  a1^4
                        2  a2  b2  a2^4
                        3  a3  b3  a3^4
```

The following call computes the mean values of the rows of the last sample:

```
>> stats::calc(%, [1, 2, 3, 4],
               (x1, x2, x3, x4) -> (x1 + x2 + x3 + x4)/4)
```

```
       1  a1  b1  a1^4  1/4*a1 + 1/4*b1 + 1/4*a1^4 + 1/4
       2  a2  b2  a2^4  1/4*a2 + 1/4*b2 + 1/4*a2^4 + 1/2
       3  a3  b3  a3^4  1/4*a3 + 1/4*b3 + 1/4*a3^4 + 3/4
```

The same is achieved by the following call:

```
>> stats::calc(%2, [1, 2, 3, 4], stats::mean)
```

```
       1  a1  b1  a1^4  1/4*a1 + 1/4*b1 + 1/4*a1^4 + 1/4
       2  a2  b2  a2^4  1/4*a2 + 1/4*b2 + 1/4*a2^4 + 1/2
       3  a3  b3  a3^4  1/4*a3 + 1/4*b3 + 1/4*a3^4 + 3/4
```

Changes:

- `stats::calc` is a new function.

`stats::ChiSquare` – the "Chi Square" distribution

`stats::ChiSquare(x, v)` computes the value

$$\int_0^x \frac{t^{v/2-1}\, e^{-t/2}}{2^{v/2}\, \Gamma(v/2)}\, dt$$

of the Chi Square distribution with v degrees of freedom at the point x.

Call(s):

- `stats::ChiSquare(x, v)`

Parameters:

`x` — an arithmetical expression.
`v` — the "degrees of freedom": a positive integer.

Return Value: an arithmetical expression.

Side Effects: The function is sensitive to the environment variable `DIGITS`, when the argument `x` is a floating point number.

Related Functions: `stats::normal`, `stats::Tdist`

Details:

- The integral is evaluated by an explicit formula.

Example 1. We compute the Chi Square distribution with one degree of freedom at the point $x = 2.4$:

```
>> stats::ChiSquare(2.4, 1)
                          0.8786647497
```

Example 2. We compute the Chi Square distribution with four degrees of freedom at a symbolic point:

```
>> stats::ChiSquare(x, 4)
                                    /   x \
                        (2 x + 4) exp| - - |
                                    \   2 /
                  1 - --------------------
                                4
```

Changes:

- No changes.

`stats::col` – select and re-arrange columns of a sample

`stats::col(s, ..)` creates a new sample from selected columns of the sample `s`.

Call(s):

- `stats::col(s, c1 <, c2, ..>)`
- `stats::col(s, c1..c2 <, c3..c4, ..>)`

Parameters:

`s`	— a sample of domain type `stats::sample`.
`c1, c2, ..`	— positive integers representing column indices of the sample `s`.

Return Value: a sample of domain type `stats::sample`.

Related Functions: `stats::concatCol`, `stats::concatRow`, `stats::row`

Details:

- `stats::col` is useful for selecting columns of interest or for re-arranging columns.
- The columns of `s` specified by the remaining arguments of `stats::col` are used to build a new sample. The new sample contains the columns of `s` in the order specified by the call to `stats::col`. Columns can be duplicated by specifying the column index more than once.

Example 1. The following sample contains columns for "gender", "age", "height", the "number of yellow socks" and "eye color" of a person:

```
>> stats::sample([["m", 26, 180, 3, "blue"],
                  ["f", 22, 160, 0, "brown"],
                  ["f", 48, 155, 2, "green"],
                  ["m", 30, 172, 1, "brown"]])

                   "m"  26  180  3  "blue"
                   "f"  22  160  0  "brown"
                   "f"  48  155  2  "green"
                   "m"  30  172  1  "brown"
```

Since nobody is really interested in the yellow socks, we create a new sample without that column:

```
>> stats::col(%, 1..3, 5)

                      "m"  26  180  "blue"
                      "f"  22  160  "brown"
                      "f"  48  155  "green"
                      "m"  30  172  "brown"
```

We can use `stats::col` to re-arrange the sample. As an illustrating example, we duplicate the first column:

```
>> stats::col(%, 1, 3, 2, 1, 4)

                  "m"  180  26  "m"  "blue"
                  "f"  160  22  "f"  "brown"
                  "f"  155  48  "f"  "green"
                  "m"  172  30  "m"  "brown"
```

Changes:

- `stats::col` is a new function.

`stats::concatCol` – concatenate samples column-wise

`stats::concatCol(s1, s2, ..)` creates a new sample consisting of the columns of the samples `s1`, `s2` etc.

Call(s):

- `stats::concatCol(s1, s2 <, s3, ..>)`

Parameters:

`s1, s2, ..` — samples of domain type `stats::sample`. Alternatively, lists may be entered, which are treated as columns of a sample.

Return Value: a sample of domain type `stats::sample`.

Related Functions: `stats::col`, `stats::concatRow`, `stats::row`

Details:

- If the samples `s1`, `s2` etc. have different numbers of rows, then the number of rows in the resulting sample is given by the "shortest" sample with the minimal number of rows. Elements below this row in "longer" samples are ignored.

Example 1. We create two samples:

```
>> s1 := stats::sample([[a1, a2], [b1, b2]]);
   s2 := stats::sample([[a3, a4], [b3, b4]])

                              a1  a2
                              b1  b2

                              a3  a4
                              b3  b4
```

Concatenation of the columns yields:

```
>> stats::concatCol(s1, s2)

                          a1  a2  a3  a4
                          b1  b2  b3  b4

>> delete s1, s2:
```

Example 2. The following sample contains columns for "gender", "age" and "height" of a person:

```
>> stats::sample([["m", 26, 180], ["f", 22, 160],
                  ["f", 48, 155], ["m", 30, 172]])

                           "m"  26  180
                           "f"  22  160
                           "f"  48  155
                           "m"  30  172
```

We append a further column "nationality", specified by a list:

```
>> stats::concatCol(%, ["German", "French", "Italian",
                        "British", "German"])

                     "m"  26  180  "German"
                     "f"  22  160  "French"
                     "f"  48  155  "Italian"
                     "m"  30  172  "British"
```

Changes:

- `stats::concatCol` is a new function.

`stats::concatRow` – **concatenate samples row-wise**

`stats::concatRow(s1, s2, ..)` creates a new sample consisting of the rows of the samples `s1`, `s2` etc.

Call(s):

- `stats::concatRow(s1, s2 <, s3, ..>)`

Parameters:

`s1, s2, ..` — samples of domain type `stats::sample`. Alternatively, lists may be entered, which are treated as rows of a sample.

Return Value: a sample of domain type `stats::sample`.

Related Functions: `stats::col`, `stats::concatCol`, `stats::row`

Details:

- All samples must have the same number of columns.

Example 1. We create a small sample:

```
>> stats::sample([[123, g], [442, f]])
```

```
                            123  g
                            442  f
```

A list is concatenated to the sample as a row:

```
>> stats::concatRow(%, [x, y])
```

```
                            123  g
                            442  f
                              x  y
```

Example 2. The following samples contain columns for "gender" and "age":

```
>> s1 := stats::sample([["f", 36], ["m", 25]]);
   s2 := stats::sample([["m", 26], ["f", 22]])
```

```
                              "f"  36
                              "m"  25

                              "m"  26
                              "f"  22
```

We build a larger sample:

```
>> stats::concatRow(s1, s2)
```

```
                              "f"  36
                              "m"  25
                              "m"  26
                              "f"  22
```

```
>> delete s1, s2:
```

Changes:

- `stats::concatRow` is a new function.

`stats::geometric` – the geometric mean

`stats::geometric(data)` returns the geometric mean of the data.

Call(s):

- `stats::geometric(x1, x2, ..)`
- `stats::geometric([x1, x2, ..])`
- `stats::geometric(s <, c>)`

Parameters:

`x1, x2, ..`	—	the statistical data: arithmetical expressions.
`s`	—	a sample of domain type `stats::sample`.
`c`	—	an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Value: an arithmetical expression.

Related Functions: `stats::a_quantil`, `stats::harmonic`, `stats::mean`, `stats::median`, `stats::modal`, `stats::quadratic`, `stats::stdev`, `stats::variance`

Details:

- The geometric mean of n data values $x_1, \ldots, x_n$ is $(x_1\, x_2 \cdots x_n)^{1/n}$.
- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.

Example 1. We calculate the geometric mean of three values:

```
>> stats::geometric(a, b, c)
```

```
                                 1/3
                          (a b c)
```

Alternatively, the data may be passed as a list:

```
>> stats::geometric([2, 3, 5])
```

```
                               1/3
                             30
```

Example 2. We create a sample:

```
>> stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
                          a1  b1  c1
                          a2  b2  c2
```

The geometric mean of the second column is:

```
>> stats::geometric(%, 2)
```

```
                                 1/2
                          (b1 b2)
```

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
                          "1996"  1242
                          "1997"  1353
                          "1998"  1142
```

We compute the geometric mean of the second column. In this case this column does not have to be specified, since it is the only non-string column in the sample:

```
>> float(stats::geometric(%))
```

```
                              1242.68722
```

Changes:

- `stats::geometric` is a new function.

`stats::harmonic` – the harmonic mean

`stats::harmonic(data)` returns the harmonic mean of the data.

Call(s):

- `stats::harmonic(x1, x2, ..)`
- `stats::harmonic([x1, x2, ..])`
- `stats::harmonic(s <, c>)`

Parameters:

`x1, x2, ..`	—	the statistical data: arithmetical expressions.
`s`	—	a sample of domain type `stats::sample`.
`c`	—	an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Value: an arithmetical expression. `FAIL` is returned, if one of the data values is zero (the harmonic mean does not exist).

Related Functions: `stats::a_quantil`, `stats::geometric`, `stats::mean`, `stats::median`, `stats::modal`, `stats::quadratic`, `stats::stdev`, `stats::variance`

Details:

- The harmonic mean of n values $x_1, \ldots, x_n$ is $n/(\frac{1}{x_1} + \cdots + \frac{1}{x_n})$.
- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.

Example 1. We calculate the harmonic mean of three values:

```
>> stats::harmonic(a, b, c)
```

$$\frac{3}{\frac{1}{a} + \frac{1}{b} + \frac{1}{c}}$$

Alternatively, the data may be passed as a list:

```
>> stats::harmonic([2, 3, 5])
```

$$90/31$$

Example 2. We create a sample:

```
>> stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

a1	b1	c1
a2	b2	c2

The harmonic mean of the second column is:

```
>> stats::harmonic(%, 2)
```

$$\frac{2}{\frac{1}{b1} + \frac{1}{b2}}$$

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
                          "1996"  1242
                          "1997"  1353
                          "1998"  1142
```

We compute the harmonic mean of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
>> float(stats::harmonic(%))
```

```
                             1239.71654
```

Changes:

- `stats::harmonic` is a new function.

`stats::kurtosis` – **kurtosis (excess)**

`stats::kurtosis(data)` returns the kurtosis (the coefficient of excess) of the data.

Call(s):

- `stats::kurtosis(x1, x2, ..)`
- `stats::kurtosis([x1, x2, ..])`
- `stats::kurtosis(s <, c>)`

Parameters:

`x1, x2, ..`	— the statistical data: arithmetical expressions.
`s`	— a sample of domain type `stats::sample`.
`c`	— an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Value: an arithmetical expression. `FAIL` is returned, if the kurtosis does not exist.

Related Functions: `stats::obliquity`

Details:

- The kurtosis of n values $x_1, \ldots, x_n$ is

$$\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^4}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^2} - 3\,,$$

  where $\bar{x}$ is the mean value of the x_i. The kurtosis measures whether a distribution is "flat" or "peaked". For normally distributed data the kurtosis is zero. If the distribution function of the data has a flatter top than the normal distribution, then the kurtosis is negative. The kurtosis is positive, if the distribution function has a high peak, compared to the normal distribution.

- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.

Example 1. We calculate the kurtosis of some values:

```
>> stats::kurtosis(0, 7, 7, 6, 6, 6, 5, 5, 4, 1)
                              -74146/271441
```

Alternatively, the data may be passed as a list:

```
>> stats::kurtosis([2, 2, 4, 6, 8, 10, 10])
                                  -85/54
```

Example 2. We create a sample:

```
>> stats::sample([[a, 5, 8], [b, 3, 7], [c, d, 0]])
                               a  5  8
                               b  3  7
                               c  d  0
```

The kurtosis of the second column is:

```
>> stats::kurtosis(%, 2)
       /       d \4     /       d \4     / 2 d       \4
     3 | 1/3 - - |  + 3 | 7/3 - - |  + 3 | --- - 8/3 |
       \       3 /      \       3 /      \  3        /
     ------------------------------------------------- - 3
     / /       d \2   /       d \2   / 2 d       \2 \2
     | | 1/3 - - |  + | 7/3 - - |  + | --- - 8/3 |  |
     \ \       3 /    \       3 /    \  3        /  /
```

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
                          "1996"  1242
                          "1997"  1353
                          "1998"  1142
```

We compute the kurtosis of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
>> stats::kurtosis(%)
```

```
                                  -3/2
```

Changes:

- `stats::kurtosis` is a new function.

`stats::linReg` – linear regression (least squares fit)

`stats::linReg(data)` returns the least squares estimators $[a, b]$ of a linear relation $y = a + b\,x$ between data pairs.

Call(s):

- `stats::linReg([x1, x2, ..], [y1, y2, ..])`
- `stats::linReg([[x1, y1], [x2, y2], ..])`
- `stats::linReg(s <, cx, cy>)`
- `stats::linReg(s <, [cx, cy]>)`

Parameters:

`x1, x2, ..`	—	statistical data: arithmetical expressions.
`y1, y2, ..`	—	statistical data: arithmetical expressions.
`s`	—	a sample of domain type `stats::sample`.
`cx, cy`	—	integers representing column indices of the sample `s`. Column `cx` provides the data `x1, x2, ..`, column `cy` provides the data `y1, y2, ...`

Return Value: a list `[a, b]` of arithmetical expressions representing the offset and the slope of the linear relation. `FAIL` is returned, if these estimators do not exist.

Related Functions: `stats::reg`, `stats::sample`

Details:

- A linear relation $y_i = a + b\,x_i + e_i$ between the data pairs (x_i, y_i) is assumed. The least squares estimators a and b are chosen such that the quadratic error $\sum e_i^2$ is minimized.
- The column indices `cx`, `cy` are optional, if the data are given by a `stats::sample` object containing only two non-string columns. Cf. example 2.
- Multivariate linear regression and non-linear regression is provided by `stats::reg`.

Example 1. We calculate the least square estimators of four pairs of values given in two lists. Note that there is a linear relation $y = 1 + 2\,x$ between the entries of the lists:

```
>> stats::linReg([0, 1, 2, 3], [1, 3, 5, 7])

                              [1, 2]
```

Alternatively, the data may be specified by a list of pairs:

```
>> stats::linReg([[0, 0], [1, 3.3], [2, 4.8], [3, 6.9]])

                           [0.42, 2.22]
```

Example 2. We create a sample consisting of one string column and two non-string columns:

```
>> stats::sample([["1", 0, 0], ["2", 10, 15], ["3", 20, 30]])

                           "1"   0   0
                           "2"  10  15
                           "3"  20  30
```

The least square estimators are calculated using the data columns 2 and 3. In this example there are only two non-string columns, so the column indices do not have to be specified:

```
>> stats::linReg(%)

                              [0, 3/2]
```

Example 3. We create a sample consisting of three data columns:

```
>> stats::sample([[1, 0, 0], [2, 10, 15], [3, 20, 30]])
```

```
                            1   0   0
                            2  10  15
                            3  20  30
```

We compute the least square estimators for the data pairs given by the first and the second column:

```
>> stats::linReg(%, 1, 2)
```

```
                              [-10, 10]
```

Example 4. We create a sample of three columns containing symbolic data:

```
>> stats::sample([[x, y, 0], [2, 4, 15], [3, 20, 30]])
```

```
                            x   y   0
                            2   4  15
                            3  20  30
```

We compute the symbolic least square estimators for the data pairs given by the first and the second column. Here we specify these columns by a list of column indices:

```
>> map(stats::linReg(%, [1, 2]), normal)
```

```
 --                            2                                    -
-
 |  13 y - 68 x - 5 x y + 24 x  - 28  2 x y - 5 y - 24 x + 84  |
 |  ------------------------------, --------------------
---  |
 |             2                              2                 |
 --         2 x  - 10 x + 14             2 x  - 10 x + 14     -
-
```

Changes:

- The input data can now be provided by a `stats::sample`.

`stats::mean` – the arithmetic mean

`stats::mean(data)` returns the arithmetic mean of the data.

Call(s):

- `stats::mean(x1, x2, ..)`
- `stats::mean([x1, x2, ..])`
- `stats::mean(s <, c>)`

Parameters:

`x1, x2, ..`	—	the statistical data: arithmetical expressions.
`s`	—	a sample of domain type `stats::sample`.
`c`	—	an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Value: an arithmetical expression.

Related Functions: `stats::a_quantil`, `stats::geometric`, `stats::harmonic`, `stats::median`, `stats::modal`, `stats::quadratic`, `stats::stdev`, `stats::variance`

Details:

- The arithmetic mean of n values $x_1, \ldots, x_n$ is $\frac{1}{n}(x_1 + \cdots + x_n)$.
- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.

Example 1. We calculate the arithmetic mean of three values:

```
>> stats::mean(a, b, c)
```

```
a   b   c
- + - + -
3   3   3
```

Alternatively, the data may be passed as a list:

```
>> stats::mean([2, 3, 5])
```

```
10/3
```

Example 2. We create a sample:

```
>> stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
a1  b1  c1
a2  b2  c2
```

The arithmetic mean of the second column is:

```
>> stats::mean(%, 2)
```

```
   b1   b2
   -- + --
   2    2
```

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

We compute the harmonic mean of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
>> float(stats::mean(%))
```

```
1245.666667
```

Changes:

- The input data can now be provided by a `stats::sample`.

`stats::meanTest` – test an estimate of an expected mean

`stats::meanTest(data, m)` returns the probability that the expected mean of the data is larger than `m`.

Call(s):

- `stats::meanTest([x1, x2, ..], m <, distribution>)`

Parameters:

`x1, x2, ..`	—	the statistical data: arithmetical expressions.
`m`	—	the estimate for the expected mean of the data: an arithmetical expression.
`distribution`	—	either `stats::normal` or `stats::Tdist`. The default is the T-distribution `stats::Tdist`.

Return Value: an arithmetical expression. `FAIL` is returned, if the variance of the data vanishes.

Related Functions: `stats::mean`, `stats::normal`, `stats::stdev`, `stats::Tdist`

Details:

- `stats::meanTest` computes $y = \sqrt{\frac{n}{v}}\,(\bar{x} - m)$, where $\bar{x} = \frac{1}{n}\,\sum_{i=1}^{n} x_i$ is the mean of the data and $v = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$ is the statistical variance.

  `stats::meanTest` returns `stats::Tdist`$(y, n-1)$ or `stats::normal`$(y, 0, 1)$, depending on the chosen distribution.

- If the data obey a normal distribution with expectation value μ, then the variable $\sqrt{\frac{n}{v}}\,(\bar{x} - \mu)$ obeys a T-distribution with $n-1$ degrees of freedom. In this case the value returned by `stats::meanTest`$(data, m)$ is the probability that $\mu \geq m$.

Example 1. 10 experiments produced the values $1, -2, 3, -4, 5, -6, 7, -8, 9, 10$. There is only a small probability that the expected mean value of the underlying distribution is larger than 5:

```
>> data := [1, -2, 3, -4, 5, -6, 7, -8, 9, 10]:
   float(stats::meanTest(data, 5))

                              0.05756660092
```

We test the hypothesis "expected mean ≥ 5" again, this time using the normal distribution:

```
>> float(stats::meanTest(data, 5, stats::normal))

                              0.04058346176

>> delete data:
```

Changes:

- No changes.

`stats::median` – the median value of discrete data

`stats::median(..)` returns the median of discrete data.

Call(s):

- `stats::median(x1, x2, ..)`
- `stats::median([x1, x2, ..])`
- `stats::median(s <, c>)`

Parameters:

`x1, x2, ..`	— the statistical data: numerical real values.
`s`	— a sample of domain type `stats::sample`.
`c`	— an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Value: an arithmetical expression. `FAIL` is returned, if the data are empty.

Related Functions: `stats::a_quantil`, `stats::geometric`, `stats::harmonic`, `stats::mean`, `stats::modal`, `stats::quadratic`, `stats::stdev`, `stats::variance`

Details:

- The median of n sorted values $x_1 \leq \cdots \leq x_n$ is $x_{(n+1)/2}$, if n is odd. It is $\frac{1}{2}\,(x_{n/2} + x_{n/2+1})$, if n is even.
- It coincides with the $\frac{1}{2}$-quantile. The call `stats::median(data)` is equivalent to the call `stats::a_quantil(1/2, data)`. See the help page of `stats::a_quantil` for details on the parameters specifying the data.

Example 1. We calculate the median of a sequence of five values:

```
>> stats::median(3, 8, 5, 9/2, 11)

                                  5
```

Alternatively, the data may be passed as a list:

```
>> stats::median([3, 8, 5, 9/2, 11])

                                  5
```

Example 2. We create a sample:

```
>> stats::sample([[4, 7, 5], [3, 6, 17], [8, 2, 2]])
```

```
4  7   5
3  6  17
8  2   2
```

The median of the second column is 6:

```
>> stats::median(%, 2)
```

```
6
```

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

The median of the second column is calculated. In this case this column does not have to be specified, since it is the only non-string column in the sample:

```
>> stats::median(%)
```

```
1242
```

Changes:

- The input data can now be provided by a `stats::sample`.

`stats::modal` – the modal (most frequent) value(s)

`stats::modal(data)` returns the most frequent value(s) of the data.

Call(s):

- `stats::modal(x1, x2, ..)`
- `stats::modal([x1, x2, ..])`
- `stats::modal(s <, c>)`

Parameters:

`x1, x2, ..`	— the statistical data: arithmetical expressions.
`s`	— a sample of domain type `stats::sample`.
`c`	— an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Value: an arithmetical expression.

Return Value: a sequence consisting of a list and an integer. The list contains the most frequent element(s) in the data, the integer specifies the number of occurrences. E.g., the result "`[data5, data10], 21`" means that `data5` and `data10` are the most frequent data items, each occurring 21 times.

Related Functions: `stats::a_quantil`, `stats::geometric`, `stats::harmonic`, `stats::mean`, `stats::median`, `stats::quadratic`, `stats::stdev`, `stats::variance`

Details:

- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.

Example 1. We calculate the modal value of a data sequence:

```
>> stats::modal(2, a, b, c, b, 10, 12, 2, b)
                                [b], 3
```

Alternatively, the data may be passed as a list:

```
>> stats::modal([a, a, a, b, c, b, 10, 12, 2, b])
                              [a, b], 3
```

Example 2. We create a sample containing "age" and "gender":

```
>> stats::sample([[32, "f"], [25, "m"], [40, "f"], [23, "f"]])
                              32  "f"
                              25  "m"
                              40  "f"
                              23  "f"
```

The modal value of the second column (the most frequent "gender") is calculated:

```
>> stats::modal(%, 2)
                              ["f"], 3
```

Example 3. We create a sample consisting of only one column:

```
>> stats::sample([4, 6, 2, 6, 8, 3, 2, 1, 7, 9, 3, 6, 5, 1, 6, 8]):
```

The modal value of these data is calculated. In this case the column does not have to be specified, since there is only one column:

```
>> stats::modal(%)
```

```
[6], 4
```

Changes:

- `stats::modal` is a new function.

`stats::normal` – the normal (Gaussian) distribution

`stats::normal(x, m, v)` computes the value

$$\frac{1}{\sqrt{2\,\pi\,v}}\,\int_{-\infty}^{x} e^{-\frac{(t-m)^2}{2\,v}}\,dt$$

of the normal distribution with mean m and variance v at the point x.

Call(s):

- `stats::normal(x <, m> <, v>)`

Parameters:

`x` — an arithmetical expression.
`m` — the mean of the distribution: an arithmetical expression.
`v` — the variance of the distribution: an arithmetical expression.

Return Value: an arithmetical expression.

Side Effects: The function is sensitive to the environment variable `DIGITS`, when the argument `x` is a floating point number.

Related Functions: `stats::ChiSquare`, `stats::Tdist`

Details:

- If the mean m and the variance v are not specified, then the default values $m = 0$, $v = 1$ are used.

Example 1. We compute the normal distribution with mean $m = 0$ and variance $v = 1$ at the point $x = 3.4$:

```
>> stats::normal(3.4)
```

```
0.9996630707
```

Example 2. We compute the normal distribution with symbolic arguments:

```
>> stats::normal(x, m, v)
```

```
                          /  1/2          \
                          | 2    (x - m) |
                      erfc| ------------ |
                          |       1/2    |
                          \    2 v       /
                  1 - --------------------
                                2
```

Changes:

- The internal representation now uses `erfc` instead of `erf`.

`stats::obliquity` – obliquity (skewness)

`stats::obliquity(data)` returns the obliquity (skewness) of the data.

Call(s):

- `stats::obliquity(x1, x2, ..)`
- `stats::obliquity([x1, x2, ..])`
- `stats::obliquity(s <, c>)`

Parameters:

`x1, x2, ..`	—	the statistical data: arithmetical expressions.
`s`	—	a sample of domain type `stats::sample`.
`c`	—	an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Value: an arithmetical expression. `FAIL` is returned, if the obliquity does not exist.

Related Functions: `stats::kurtosis`

Details:

- The obliquity of n values $x_1, \ldots, x_n$ is

$$\frac{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^3}{\left(\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2\right)^{\frac{3}{2}}},$$

  where $\bar{x}$ is the mean value of the x_i. The obliquity is a measure for the symmetry of a distribution. It is zero, if the distribution of the data is symmetric around the mean. Positive values indicate that the distribution function has a "longer tail" to the right of the mean than to the left. Negative values indicate a "longer tail" to the left.

- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.

Example 1. We calculate the obliquity of a data sequence:

```
>> float(stats::obliquity(0, 7, 7, 6, 6, 6, 5, 5, 4, 1))
                              -1.041368312
```

Alternatively, the data may be passed as a list:

```
>> stats::obliquity([2, 2, 4, 6, 8, 10, 10])
                                    0
```

Example 2. We create a sample:

```
>> stats::sample([[a, 5, 8], [b, 3, 7], [c, d, 0]])
                              a  5  8
                              b  3  7
                              c  d  0
```

The obliquity of the second column is:

```
>> stats::obliquity(%, 2)
       1/2 / /       d \3   /       d \3   / 2 d       \3 \
      3    | | 1/3 - - |  + | 7/3 - - |  + | --- - 8/3 |  |
           \ \       3 /    \       3 /    \  3        /  /
      ------------------------------------------------------
-
       / /       d \2   /       d \2   / 2 d       \2 \3/2
       | | 1/3 - - |  + | 7/3 - - |  + | --- - 8/3 |  |
       \ \       3 /    \       3 /    \  3        /  /
```

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

We compute the obliquity of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
>> float(stats::obliquity(%))
```

```
0.06374333648
```

Changes:

- `stats::obliquity` is a new function.

`stats::quadratic` – the quadratic mean

`stats::quadratic(data)` returns the quadratic mean of the data.

Call(s):

- `stats::quadratic(x1, x2, ..)`
- `stats::quadratic([x1, x2, ..])`
- `stats::quadratic(s <, c>)`

Parameters:

`x1, x2, ..`	—	the statistical data: arithmetical expressions.
`s`	—	a sample of domain type `stats::sample`.
`c`	—	an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Return Value: an arithmetical expression.

Related Functions: `stats::a_quantil`, `stats::geometric`, `stats::harmonic`, `stats::mean`, `stats::median`, `stats::modal`, `stats::stdev`, `stats::variance`

Details:

- The quadratic mean of n values $x_1, \ldots, x_n$ is $\sqrt{\frac{1}{n}\sum_{i=1}^{n} {x_i}^2}$.
- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.

Example 1. We calculate the quadratic mean of three values:

```
>> stats::quadratic(a, b, c)
```

```
                        /  2    2    2 \1/2
                        | a    b    c  |
                        | -- + -- + -- |
                        \ 3    3    3  /
```

Alternatively, the data may be passed as a list:

```
>> stats::quadratic([2, 3, 5])
```

```
                             1/2   1/2
                            3    38
                            ----------
                                3
```

Example 2. We create a sample:

```
>> stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
                           a1  b1  c1
                           a2  b2  c2
```

The quadratic mean of the second column is:

```
>> stats::quadratic(%, 2)
```

```
                          /   2     2 \1/2
                          | b1    b2  |
                          | --- + --- |
                          \  2     2  /
```

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
"1996"  1242
"1997"  1353
"1998"  1142
```

We compute the quadratic mean of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
>> float(stats::quadratic(%))
```

```
1248.644198
```

Changes:

- `stats::quadratic` is a new function.

`stats::reg` – regression (general least square fit)

Consider a "model function" f with n parameters $p_1, \ldots, p_n$ relating a dependent variable y and m independent variables $x_1, \ldots, x_m$:

$$y = f(x_1, \ldots, x_m; p_1, \ldots, p_n) \, .$$

Given k different measurements $x_{1j}, \ldots, x_{kj}$ for the independent variables x_j and corresponding measurements $y_1, \ldots, y_k$ for the dependent variable y, one fits the parameters $p_1, \ldots, p_n$ by minimizing the "weighted quadratic deviation"

$$\Delta^2(p_1, \ldots, p_n) = \sum_{i=1}^{k} w_i \, |y_i - f(x_{i1}, \ldots, x_{im}; p_1, \ldots, p_n)|^2 \, .$$

`stats::reg(..data.., f,..)` computes numerical approximations of the fit parameters.

Call(s):

```
stats::reg([x.1.1,..,x.k.1], .., [x.1.m,..,x.k.m],
           [y.1,..,y.k] <, [w.1,..,w.k]>, f,
           [x.1,..,x.m], [p.1,..,p.n] <, Starting-
           Values = [p.1(0),..,p.n(0)]>)
```

```
stats::reg([[x.1.1,..,x.1.m, y.1 <, w.1>], ..,
            [x.k.1,..,x.k.m, y.k <, w.k>]], f,
            [x.1,..,x.m], [p.1,..,p.n] <, Starting-
            Values = [p.1(0),..,p.n(0)]>)
stats::reg(s, c.1, .., c.m, cy <, cw>, f,
            [x.1,..,x.m], [p.1,..,p.n] <, Starting-
            Values = [p.1(0),..,p.n(0)]>)
stats::reg(s, [c.1, .., c.m, cy <, cw>], f,
            [x.1,..,x.m], [p.1,..,p.n] <, Starting-
            Values = [p.1(0),..,p.n(0)]>)
```

Parameters:

`x.1.1,..,x.k.m` — numerical sample data for the independent variables. The entry `x.i.j` represents the *i*-th measurement of the independent variable `x.j`.

`y.1,..,y.k` — numerical sample data for the dependent variable. The entry `y.i` represents the *i*-th measurement of the dependent variable.

`w.1,..,w.k` — numerical weight factors. The entry `w.i` is used as a weight for the data `(x.i.1,..,x.i.m,y.i)` of the *i*-th measurement. If no weights are provided, then `w.i = 1` is used.

`f` — the model function: an arithmetical expression representing a function of the independent variables `x.1, .., x.m` and the fit parameters `p.1, .., p.n`. The expression must not contain any symbolic objects apart from `x.1, .., p.n`.

`x.1,..,x.m` — the independent variables: identifiers or indexed identifiers.

`p.1,..,p.n` — the fit parameters: identifiers or indexed identifiers.

`p.1(0),..,p.n(0)` — The user can assist the internal numerical search by providing numerical starting values `p.i(0)` for the fit parameters `p.i`. These should be reasonably close to the optimal fit values. The starting values `p.i(0) = 1.0` are used, if no other values are provided by the user.

`s` — a sample of domain type `stats::sample` containing the data `x.i.j` for the independent variables, the data `y.i` for the dependent variable and, optionally, the weights `w.i`.

`c.1,..,c.m` — positive integers representing column indices of the sample `s`. Column `c.j` provides the measurements `x.i.j` for the independent variable `x.j`.

`cy` — a positive integer representing a column index of the sample `s`. This column provides the measurements `y.i` for the dependent variable.

`cw` — a positive integer representing a column index of the sample `s`. This column provides the weight factors `w.i`.

Return Value: a list $[[p_1, \ldots, p_n], \Delta^2]$ containing the optimized fit parameters p_i minimizing the quadratic deviation. The minimized value of this deviation is given by Δ^2, it indicates the quality of the fit. All returned data are floating point values. `FAIL` is returned, if a least square fit of the data is not possible with the given model function or if the internal numerical search failed.

Side Effects: The function is sensitive to the environment variable `DIGITS`, which determines the numerical working precision.

Related Functions: `stats::linReg`, `stats::sample`

Details:

- All data must be convertible to real or complex floating point values via `float`.
- The number of measurements k must not be less than the number n of parameters p_i.
- The model function f may be non-linear in the independent variables x_i and the fit parameters p_i. E.g., a model function such as `p1 + p2*x1^2 + exp(p3 + p4*x2)` with the independent variables `x1, x2` and the fit parameters `p1, .., p4` is accepted.
- Note that the fitting of model functions with a non-linear dependence on the parameters p_i is much more costly than a linear regression, where the p_i enter linearly. The functional dependence of the model function on the variables x_i is of no relevance.
- There are rare cases where the implemented algorithm converges to a local minimum rather than to a global minimum. In particular, this problem may arise when the model involves periodic functions. It is recommended to provide suitable starting values for the fit parameters in this case. Cf. example 4.

Option <`StartingValues = [p.1(0),..,p.n(0)]`>:

- If the model function depends linearly on the fit parameters p_i ("linear regression"), then the optimized parameters are the solution of a linear system of equations. In this case there is no need to provide starting values for a numerical search. In fact, initial values provided by the user are ignored.
- If the model function depends non-linearly on the fit parameters p_i ("non-linear regression"), then the optimized fitting parameters are the solution of a non-linear optimization problem. There is no guarantee that the internal search for a numerical solution will succeed. It is recommended to

assist the internal solver by providing reasonably good estimates for the optimal fit parameters.

Example 1. We fit a linear function $y = p_1 + p_2\,x_1$ to four data pairs (x_{i1}, y_i) given by two lists:

```
>> stats::reg([0, 1, 2, 3], [1, 3, 5, 7],
              p1 + p2*x1, [x1], [p1, p2])

                          [[1.0, 2.0], 0.0]
```

The parameter values $p_1 = 1.0$, $p_2 = 2.0$ provide a perfect fit: the quadratic deviation vanishes.

Example 2. We fit an exponential function $y = a\,e^{b\,x}$ to five data pairs (x_i, y_i). Weights are used to decrease the influence of the "exceptional pair" $(x, y) = (5.0, 6.5 \times 10^6)$ on the fit:

```
>> stats::reg([[1.1, 54, 1], [1.2, 73, 1], [1.3, 98, 1],
               [1.4, 133, 1], [5.0, 6.5*10^6, 10^(-4)]],
              a*exp(b*x), [x], [a, b])

           [[1.992321622, 2.999602426], 0.2001899629]
```

Example 3. We create a sample with four columns. The first column is a counter labeling the measurements. The second and third column provide measured data of two variables x_1 and x_2, respectively. The last column provides corresponding measurements of a dependent variable.

```
>> s := stats::sample([[1, 0, 0,  1.1], [2, 0, 1,  5.4],
                       [3, 1, 1,  8.5], [4, 1, 2, 18.5],
                       [5, 2, 1, 15.0], [6, 2, 2, 24.8]])

                         1  0  0   1.1
                         2  0  1   5.4
                         3  1  1   8.5
                         4  1  2  18.5
                         5  2  1  15.0
                         6  2  2  24.8
```

First, we try to model the data provided by the columns 2, 3, 4 by a function that is linear in the variables x_1, x_2. We specify the data columns by a list of column indices:

```
>> stats::reg(s, [2, 3, 4], p1 + p2*x1 + p3*x2,
              [x1, x2], [p1, p2, p3])
```

```
[[-0.9568181818, 4.688636364, 7.272727273], 15.23613636]
```

The quadratic deviation is rather large, indicating that a linear function is inappropriate to fit the data. Next, we extend the model and consider a polynomial fit function of degree 2. This is still a linear regression problem, because the fit parameters enter the model function linearly. We specify the data columns by a sequence of column indices:

```
>> stats::reg(s, 2, 3, 4,
              p1 + p2*x1 + p3*x2 + p4*x1^2 + p5*x2^2,
              [x1, x2], [p1, p2, p3, p4, p5])

              [[1.1, 1.525, 1.5, 1.625, 2.8], 0.01]
```

Finally, we include a further term `p6*x1*x2` in the model, obtaining a perfect fit:

```
>> stats::reg(s, 2, 3, 4,
              p1 + p2*x1 + p3*x2 + p4*x1^2 + p5*x2^2 + p6*x1*x2,
              [x1, x2], [p1, p2, p3, p4, p5, p6])

      [[1.1, 1.6, 1.35, 1.7, 2.95, -0.2], 4.267632241e-34]

>> delete s:
```

Example 4. We create a sample of two columns:

```
>> s := stats::sample([[1, -1.44], [2, -0.82],
                       [3, 0.97], [4, 1.37]])

                           1  -1.44
                           2  -0.82
                           3   0.97
                           4   1.37
```

The data are to be modeled by a function of the form $y = p_1 \sin(p_2\, x)$, where the first column contains measurements of x and the second column contains corresponding data for y. Note that in this example there is no need to specify column indices, because the sample contains only two columns:

```
>> stats::reg(s, a*sin(b*x), [x], [a, b])

         [[-1.499812823, 1.281963381], 0.00001255632629]
```

Fitting a periodic function may be problematic. We provide starting values for the fit parameters and obtain a quite different set of parameters approximating the data with the same quality:

```
>> stats::reg(s, a*sin(b*x), [x], [a, b], StartingValues = [2, 5])

         [[1.499812823, 5.001221926], 0.00001255632629]

>> delete s:
```

Background:

- `stats::reg` uses a Marquard-Levenberg gradient expansion algorithm. Searching for the minimum of $\Delta^2(p_1, \ldots, p_n)$ the algorithm does not simply follow the negative gradient, but the diagonal terms of the curvature matrix are increased by a factor that is optimized in each step of the search.

Changes:

- `stats::reg` is a new function.

`stats::row` – select and re-arrange rows of a sample

`stats::row(s, ..)` creates a new sample from selected rows of the sample `s`.

Call(s):

- `stats::row(s, r1 <, r2, ..>)`
- `stats::row(s, r1..r2 <, r3..r4, ..>)`

Parameters:

`s`	—	a sample of domain type `stats::sample`.
`r1, r2, ..`	—	positive integers representing row indices of the sample `s`.

Return Value: a sample of domain type `stats::sample`.

Related Functions: `stats::col`, `stats::concatCol`, `stats::concatRow`, `stats::selectRow`

Details:

- `stats::row` is useful for selecting rows of interest or for re-arranging rows.
- The rows of `s` specified by the remaining arguments of `stats::row` are used to build a new sample. The new sample contains the rows of `s` in the order specified by the call to `stats::row`. Rows can be duplicated by specifying the row index more than once.

Example 1. The following sample represents the "population" of a small town:

```
>> stats::sample([["1990", 10564], ["1991", 10956],
                  ["1992", 11007], ["1993", 11123],
                  ["1994", 11400], ["1995", 11645]])
```

```
                        "1990"  10564
                        "1991"  10956
                        "1992"  11007
                        "1993"  11123
                        "1994"  11400
                        "1995"  11645
```

We are only interested in the years 1990, 1991, 1992 and 1995. We create a new sample containing the rows of interest:

```
>> stats::row(%, 1..3, 6)
```

```
                        "1990"  10564
                        "1991"  10956
                        "1992"  11007
                        "1995"  11645
```

We reorder the sample:

```
>> stats::row(%, 4, 3, 2, 1)
```

```
                        "1995"  11645
                        "1992"  11007
                        "1991"  10956
                        "1990"  10564
```

Changes:

- `stats::row` is a new function.

`stats::sample` – the domain of statistical samples

A *sample* represents a collection of statistical data, organized as a matrix. Usually, each row refers to an individual of the population described by the sample. Each column represents an attribute.

`stats::sample([[a11, ..., a1.n], ..., [a.m.1, ..., a.m.n]])`
creates a sample with m rows and n columns, `a.i.j` being the entry in the i-th row, j-th column.

`stats::sample([a11, ..., a.m.1])` creates a sample with m rows and one column.

Creating Elements:

- `stats::sample([[a11, a12, ...], [a21, a22, ...], ...])`
- `stats::sample([a11, a21, ...])`

Parameters:

`a11, a12, ...` — arithmetical expressions or strings.

Categories:

`Cat::Set`

Details:

- Each row `[a.i.1, ..., a.i.n]` must contain the same number of entries.
- Elements of domain type `DOM_COMPLEX`, `DOM_EXPR`, `DOM_FLOAT`, `DOM_IDENT`, `DOM_INT`, or `DOM_RAT` are regarded as "data" and are stored in a sample as on input. All other types of input parameters are converted to strings (`DOM_STRING`).

  If one element in a column is a string or is converted to a string, then all elements of that column are converted to strings.

  This produces two kinds of columns: data columns and string columns.

Mathematical Methods

Method `equal`: test for equality

`equal(`*`dom`* `s1, `*`dom`* `s2)`

- tests, whether the two samples `s1` and `s2` are equal. Returns `TRUE` or `FALSE`, respectively.

Conversion Methods

Method `convert`: convert a list to a sample

`convert(`*`list`* `x )`

- converts the list/list of lists `x` to a sample. Returns `FAIL`, if this is not possible.

Method `convert_to`: convert a sample to a list of lists

```
convert_to(dom s, type T)
```

- Presently, only `T = DOM_LIST` is implemented; a list of all rows of `s` is returned as a list of lists. All other target types `T` yield `FAIL`.

Method `expr`: convert a sample to a list of lists of expressions

```
expr(dom s)
```

- converts `s` to a list of lists. All entries are converted to expressions.

Access Methods

Method `col2list`: return a particular column as a list

```
col2list(dom s, positive integer or range c, ...)
```

- returns the `c`-th column of the sample `s` as a list. It is possible to specify more than one column index or range of column indices.

Method `append`: append a row

```
append(dom s, list row)
```

- appends the list `row` as a row to the sample `s`. The length of the row has to coincide with the number of columns of the sample `s`.

Method `_concat`: create a sample from the rows of several samples

```
_concat(dom s, dom or list s1, ...)
```

- returns a sample consisting of the rows of `s` and the rows of the further arguments. Lists are regarded as samples with one row. All rows must have the same length.
  Note that the dot operator `.` may be used to call this method. Cf. example 4.
- This method overloads `_concat`.

Method `delCol`: delete one or more columns

```
delCol(dom s, positive integer or range c, ...)
```

- returns the sample obtained by deleting the column(s) of the sample `s` specified by the argument `c`. `NIL` is returned, if all columns of `s` are deleted.
  It is possible to specify more than one column index or range of column indices.

Method `delRow`: delete one or more rows

```
delRow(dom s, positive integer or range r, ...)
```

- returns the sample obtained by deleting the row(s) of the sample `s` specified by the argument `r`. `NIL` is returned, if all rows of `s` are deleted.

  It is possible to specify more than one row index or range of row indices.

Method `float`: map the float function to all entries

```
float(dom s)
```

- applies `float` to all entries of `s`.
- This method overloads `float`.

Method `has`: test for the occurrence of elements

```
has(dom s, list or set or expression e)
```

- tests, whether an expression `e` is among the entries of `s`. Returns `TRUE` or `FALSE`, respectively.
- If `e` is a list or a set, then this method tests, whether at least one of its elements is among the entries of `s`.

Method `_index`: return a particular entry

```
_index(dom s, positive integer i, positive integer j)
```

- returns the `j`-th entry of the `i`-th row of the sample `s`.
- This method overloads `_index`.
- Indexed calls such as `s[i, j]` call this method.

Method `set_index`: assign a new value to an entry

```
set_index(dom s, positive integer i, positive integer
j, any x)
```

- sets the (`i`,`j`)-th element of `s` to `x`.

  Note that no conversion to strings occurs, even if the type of `s` is not one of the "data types" described in the 'Details' section.

- This method is called by indexed assignments of the form `s[i, j] := x`.

Method `map`: map a function to the rows

```
map(dom s, any f)
```

- maps the function `f` onto the rows of the sample `s`. Note that rows are internally represented by lists. The function must accept a list as input parameter and must return a list of the same length.
- This method overloads `map`.

Method `nops`: number of rows

```
nops(dom s)
```

- returns the number of rows of `s`.
- This method overloads `nops`.

Method `op`: get the operands (rows)

```
op(dom s, positive integer i )
op(dom s, [positive integer i, positive integer j])
```

- returns the *i*-th row of `s` or the *j*-th element of the *i*-th row, respectively.
- This method overloads `op`.

Method `subsop`: replace a row

```
subsop(dom s, integer i = list newrow, ..)
```

- replaces the *i*-th row of the sample `s` by `newrow`. The length of the new row has to match the number of columns of `s`. It is possible to replace several rows simultaneously.
- This method overloads `subsop`.

Method `row2list`: return a particular row as a list

```
row2list(dom s, positive integer or range r, ..)
```

- returns the `r`-th row of the sample `s` as a list. It is possible to specify more than one row index or range of row indices.

Technical Methods

Method `print`: output

```
print(dom s)
```

- returns a matrix-like scheme containing the entries of the sample `s`. This method is called by the system for displaying samples.
- This method overloads `print`.

Method `fastprint`: fast output

`fastprint(`*dom* `s)`

- returns a matrix-like scheme containing the entries of the sample `s`. The usual `print` command may be slow for large samples. This method provides a somewhat faster alternative.

Example 1. A sample is created from a list of rows:

```
>> stats::sample([[5, a], [b, 7.534], [7/4, c+d]])
```

```
    5      a
    b  7.534
  7/4  c + d
```

For a sample with only one column one can use a flat list instead of a list of rows:

```
>> stats::sample([5, 3, 8])
```

```
5
3
8
```

Example 2. The following input creates a small sample with columns for "gender", "age" and "height", respectively:

```
>> stats::sample([["m", 26, 180], ["f", 22, 160],
                  ["f", 48, 155], ["m", 30, 172]])
```

```
  "m"  26  180
  "f"  22  160
  "f"  48  155
  "m"  30  172
```

Note that all entries in a column are automatically converted to strings, if one entry of that column is a string:

```
>> stats::sample([[m, 26, 180], [f, 22, 160],
                  ["f", 48, 155], [m, 30, 172]])
```

```
  "m"  26  180
  "f"  22  160
  "f"  48  155
  "m"  30  172
```

Example 3. The functions `float`, `has`, `map`, `nops`, `op`, and `subsop` are overloaded to work on samples as on lists of lists:

```
>> s := stats::sample([[a, 1], [b, 2], [c, 3]])

                              a  1
                              b  2
                              c  3

>> float(s), has(s, a), map(s, list -> [list[1], list[2]^2]),
   nops(s), subsop(s, 1 = [d, 4]), op(s, [1, 2])

             a  1.0  , TRUE, a  1  , 3, d  4  , 1
             b  2.0          b  4       b  2
             c  3.0          c  9       c  3
```

Indexing works like on `array`s:

```
>> s[1, 2] := x : s

                              a  x
                              b  2
                              c  3

>> delete s:
```

Example 4. The dot operator may be used to concatenate samples and lists (regarded a samples with one row):

```
>> s := stats::sample([[1, a], [2, b]]): s.[X, Y].s

                              1  a
                              2  b
                              X  Y
                              1  a
                              2  b

>> delete s:
```

Super-Domain: `Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Changes:

- `stats::sample` is a new function.

`stats::sample2list` – convert a sample to a list of lists

`stats::sample2list(s)` converts the sample `s` to a list of lists.

Call(s):

- `stats::sample2list(s)`

Parameters:

`s` — a sample of domain type `stats::sample`.

Return Value: a list of lists.

Related Functions: `stats::unzipCol`, `stats::zipCol`

Details:

- The sub-lists of the list returned by `stats::sample2list(s)` are the rows of the sample `s`.

Example 1. First we create a sample from a list of lists:

```
>> stats::sample([[123, s, 1/2], [442, s, -1/2], [322, p, -
1/2]])
```

```
                        123  s   1/2
                        442  s  -1/2
                        322  p  -1/2
```

The input list may be recovered by `stats::sample2list`:

```
>> stats::sample2list(%)
```

```
         [[123, s, 1/2], [442, s, -1/2], [322, p, -1/2]]
```

Changes:

- `stats::sample2list` is a new function.

`stats::selectRow` – select rows of a sample

`stats::selectRow(s, ..)` selects rows of the sample `s` having specific entries in specific places.

Call(s):

- `stats::selectRow(s, c, x <, Not>)`
- `stats::selectRow(s, [c1, c2, ..], [x1, x2, ..] <, Not>)`

Parameters:

`s`	—	a sample of domain type `stats::sample`.
`c, c1, c2, ..`	—	integers representing column indices of the sample `s`.
`x, x1, x2, ..`	—	arithmetical expressions.

Options:

`Not` — causes `stats::selectRow` to select those rows which do *not* have the specified entries.

Return Value: a sample of domain type `stats::sample`.

Related Functions: `stats::row`

Details:

- `stats::selectRow(s, c, x)` returns a sample consisting of all rows in `s`, which contain the data element `x` at the position `c`.
- `stats::selectRow(s, [c1, c2, ..], [x1, x2, ..])` returns a sample consisting of all rows in `s`, which contain the data element `x1` at the position `c1` *and* `x2` at the position `c2` etc. There must be as many positions `c1`, `c2`, ... as data elements `x1`, `x2`, ...

Example 1. We create a sample with two columns:

```
>> stats::sample([[a, 5], [c, 1], [a, 2], [b, 3]])
                              a  5
                              c  1
                              a  2
                              b  3
```

We select all rows with `a` as their first entry:

```
>> stats::selectRow(%, 1, a)
                              a  5
                              a  2
```

Example 2. We create a sample containing income and costs in the years 1997 and 1998:

```
>> stats::sample([[123, "costs", "97"], [442, "income", "98"],
                  [11, "costs", "98"], [623, "income", "97"]])
                      123  "costs"   "97"
                      442  "income"  "98"
                       11  "costs"   "98"
                      623  "income"  "97"
```

We select the row which has `"income"` in the second and `"97"` in the third column:

```
>> stats::selectRow(%, [2, 3], ["income", "97"])
                      623  "income"  "97"
```

We select the remaining rows:

```
>> stats::selectRow(%2, [2, 3], ["income", "97"], Not)
                      123  "costs"   "97"
                      442  "income"  "98"
                       11  "costs"   "98"
```

Changes:

- `stats::selectRow` is a new function.

`stats::sortSample` – sort the rows of a sample

`stats::sortSample(s, ..)` sorts the rows of the sample `s`.

Call(s):

- `stats::sortSample(s)`
- `stats::sortSample(s, c1, c2, ..)`
- `stats::sortSample(s, [c1, c2, ..])`

Parameters:

`s`	— a sample of domain type `stats::sample`.
`c1, c2, ..`	— integers representing column indices of the sample `s`.

Return Value: a sample of domain type `stats::sample`.

Related Functions: `stats::selectRow`

Details:

- The sorting of rows only uses the entries of the specified columns. First, rows are sorted according to the elements of the first specified column. Those rows with identical elements in the first specified column are then ordered according to the elements in the second specified column etc.
- If no columns are specified, then column 1 is used for sorting. In case of a tie, column 2 is used etc.
- Numbers are sorted numerically, strings are sorted lexicographically. Identifiers are sorted according to the strategy used by MuPAD's `sort` command. Identifiers come first, numbers second.

Example 1. We create a sample with one column and sort it:

```
>> stats::sortSample(stats::sample([x, g2, 3, g1, 8/5, 2]))
```

```
   x
  g1
  g2
 8/5
   2
   3
```

Example 2. We create a sample with two columns:

```
>> stats::sample([[b, 2], [a, 5], [a, 2], [c, 1], [b, 3]])
```

```
b  2
a  5
a  2
c  1
b  3
```

Note the different sorting priorities specified by the column indices:

```
>> stats::sortSample(%, 1), stats::sortSample(%, 2),
   stats::sortSample(%, 1, 2), stats::sortSample(%, 2, 1)
```

```
a  2  , c  1  , a  2  , c  1
a  5    a  2    a  5    a  2
b  3    b  2    b  2    b  2
b  2    b  3    b  3    b  3
c  1    a  5    c  1    a  5
```

Example 3. We create a sample containing income and costs in the years 1997 and 1998:

```
>> stats::sample([[123, "costs", "97"], [720, "income", "98"],
                  [623, "income", "97"], [150, "costs", "98"]])
```

```
123  "costs"   "97"
720  "income"  "98"
623  "income"  "97"
150  "costs"   "98"
```

We sort according to the year (third column):

```
>> stats::sortSample(%, 3)
```

```
623  "income"  "97"
123  "costs"   "97"
150  "costs"   "98"
720  "income"  "98"
```

We sort with priority on the year. Items of the same year are then sorted lexicographically ("costs" before "income"):

```
>> stats::sortSample(%2, 3, 2)
```

```
123  "costs"   "97"
623  "income"  "97"
150  "costs"   "98"
720  "income"  "98"
```

Changes:

- `stats::sortSample` is a new function.

`stats::stdev` – the standard deviation

`stats::stdev(data)` returns the standard deviation of the data.

Call(s):

- `stats::stdev(x1, x2, .. <, Sample>)`
- `stats::stdev([x1, x2, ..] <, Sample>)`
- `stats::stdev(s <, c> <, Sample>)`

Parameters:

`x1, x2, ..`	— the statistical data: arithmetical expressions.
`s`	— a sample of domain type `stats::sample`.
`c`	— an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Options:

`Sample`	— with this option the given data are regarded as a "sample", not as a full population.

Return Value: an arithmetical expression.

Related Functions: `stats::a_quantil`, `stats::geometric`, `stats::harmonic`, `stats::mean`, `stats::median`, `stats::modal`, `stats::quadratic`, `stats::variance`

Details:

- The standard deviation of n values $x_1, \ldots, x_n$ is $\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$, where $\bar{x}$ is the arithmetic mean of the x_i. It is the square-root of the variance. With the option *`Sample`* the value $\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$ is returned instead.
- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.

Example 1. We calculate the standard deviation of three values:

```
>> stats::stdev(2, 3, 5)
```

$$\frac{\sqrt{14}}{3}$$

Alternatively, the data may be passed as a list:

```
>> stats::stdev([2, 3, 5])
```

$$\frac{\sqrt{14}}{3}$$

Example 2. We create a sample:

```
>> stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

a1	b1	c1
a2	b2	c2

The standard deviation of the second column is:

```
>> expand(stats::stdev(%, 2))
```

$$\left| \frac{b1}{4} - \frac{b1\ b2}{2} + \frac{b2}{4} \right|^{1/2}$$

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

"1996"	1242
"1997"	1353
"1998"	1142

We compute the standard deviation of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
>> float(stats::stdev(%))
```

```
                                86.17939945
```

We repeat the computation with the option *Sample*:

```
>> float(stats::stdev(%2, Sample))
                                105.5477775
```

Changes:

- The input data can now be provided by a `stats::sample`.

`stats::tabulate` – statistics of duplicate rows

`stats::tabulate(s)` eliminates duplicate rows in the sample `s` and appends a column containing the multiplicities.

`stats::tabulate(s, c1, c2, ..., f)` combines all rows that are identical except for entries in the specified columns `c1`, `c2` etc. The function `f` is applied to these columns, its result replaces the values in these columns.

`stats::tabulate(s, [c1, f1], [c2, f2], ...)` combines all rows that are identical except for entries in the columns `c1`, `c2` etc. The functions `f1`, `f2` etc. are applied to these columns, the results replace the values in these columns.

Call(s):

- `stats::tabulate(s)`
- `stats::tabulate(s, c1, c2, ... <, f>)`
- `stats::tabulate(s, c1..c2, c3..c4, ... <, f>)`
- `stats::tabulate(s, [c1, f1], [c2, f2], ...)`
- `stats::tabulate(s, [c1, c2, ..., f1], [c3, c4, ..., f2], ...)`

Parameters:

`s`	— a sample of domain type `stats::sample`
`c1, c2, ...`	— integers representing column indices of the sample `s`
`f, f1, f2, ...`	— procedures

Return Value: a sample of domain type `stats::sample`.

Related Functions: `stats::calc`

Details:

- `stats::tabulate` regards rows as duplicates, if they have identical entries in the columns that are *not* specified.
- With `stats::tabulate(s, c1, c2, ..., f)` the function `f` is applied to the entries of the duplicate rows in the specified columns. Duplicates are eliminated and replaced by a single instance of the row, the result of `f` is inserted into the corresponding columns.

  The function `f` must accept as many parameters as there are duplicates. Typical applications involve functions such as `stats::mean` which accept arbitrarily many arguments.

  E.g., with `stats::mean` duplicate rows are replaced by a single row, in which the entries of the columns `c1`, `c2` etc. are replaced by the mean values of the corresponding entries of the duplicates.

  If no function `f` is specified, then the default function `_plus` is used.

  If column indices are specified more than once, then extra columns with the result of the specified function are inserted into the sample.
- Consecutive columns may be specified by ranges. E.g., the call

  `stats::tabulate(s, c1..c2, ..., f)`

  is a short hand notation for

  `stats::tabulate(s, c1, c1+1, ..., c2, ..., f)`.
- With `stats::tabulate(s, [c1, f1], [c2, f2], ...)` pairs of columns and corresponding procedures are specified. Again, rows are regarded as duplicates, if they have identical entries in the columns that are *not* specified. Duplicates are eliminated and replaced by a single instance of the row, the result of `f1` is inserted in column `c1`, the result of `f2` is inserted in column `c2` etc.

  If column indices are specified more than once, then extra columns with the result of the specified functions are inserted into the sample.
- With `stats::tabulate(s, [c1, c2, ..., f1], ...)` it is possible to apply functions that act on several columns. The procedure `f1` has to accept a sequence of lists (each representing a column). The specified columns are replaced by a single column containing the result of `f1`. If column indices are specified more than once, then extra columns with the result of the specified function(s) are inserted into the sample. Cf. examples 2 and 3.

Example 1. We create a sample:

```
>> s := stats::sample([[a, A, 1], [a, A, 1], [a, A, 2],
                      [b, B, 5], [b, B, 10]])

                          a  A   1
                          a  A   1
                          a  A   2
                          b  B   5
                          b  B  10
```

Duplicate rows of the sample are counted. There are four unique rows, one occurring twice:

```
>> stats::tabulate(s)

                         a  A   1  2
                         a  A   2  1
                         b  B   5  1
                         b  B  10  1
```

In the following call rows are regarded as duplicates, if the entries in the first two columns coincide. We compute the mean value of the third entry of the duplicates:

```
>> stats::tabulate(s, 3, stats::mean)

                         a  A   4/3
                         b  B  15/2
```

We compute both the mean and the standard deviation of the data in the third column for the sub-samples labeled 'a A' and 'b B' by the first two columns:

```
>> stats::tabulate(s, [3, stats::mean], [3, stats::stdev])

                   a  A   4/3  1/3*2^(1/2)
                   b  B  15/2          5/2

>> delete s:
```

Example 2. We create a sample containing columns for "gender", "age" and "size":

```
>> s := stats::sample([["f", 25, 166], ["m", 30, 180],
                      ["f", 54, 160], ["m", 40, 170],
                      ["f", 34, 170], ["m", 20, 172]])
```

```
                    "f"  25  166
                    "m"  30  180
                    "f"  54  160
                    "m"  40  170
                    "f"  34  170
                    "m"  20  172
```

We use `stats::mean` on the second and third column to calculate the average "age" and "size" of each gender:

```
>> stats::tabulate(s, 2..3, float@stats::mean)

                "f"  37.66666667  165.3333333
                "m"         30.0        174.0
```

With the next call both the mean and the standard deviation of "age" and "size" for each gender are inserted into the sample.

```
>> stats::tabulate(s,
     [2, float@stats::mean], [2, float@stats::stdev],
     [3, float@stats::mean], [3, float@stats::stdev])

    "f"  37.66666667  12.11977264  165.3333333  4.109609335
    "m"         30.0  8.164965809        174.0  4.320493799
```

We compute the Bravais-Pearson correlation coefficient between "age" and "size" for each gender:

```
>> stats::tabulate(s, [2, 3, float@stats::BPCorr])

                    "f"  -0.7540135992
                    "m"  -0.1889822365

>> delete s:
```

Example 3. We create a sample:

```
>> s := stats::sample([[a, x1, 1, 2], [b, x2, 2, 4],
                      [b, x1, 2, 4], [e, x2, 3, 5.5]])

                        a  x1  1    2
                        b  x2  2    4
                        b  x1  2    4
                        e  x2  3  5.5
```

We regard rows with the same entry in the second column as "of the same kind". We tabulate the sample using different functions on the remaining columns:

```
>> stats::tabulate(s, [1, _plus], [3, _mult], [4, stats::mean])

                      a + b  x1  2     3
                      b + e  x2  6  4.75
```

One can apply customized procedures. In the following we define the procedure `plusmult`, which sums up the elements of two lists (representing columns) and then multiplies the sums.

```
>> plusmult := proc(x, y) begin _plus(op(x))*_plus(op(y)) end_proc:
```

This procedure is then used to combine the first and the third column. Simultaneously, the mean and the standard deviation of the fourth column is inserted into the sample.

```
>> stats::tabulate(s, [1, 3, plusmult], [4, stats::mean],
                   [4, stats::stdev])

                   3*a + 3*b  x1     3     1
                   5*b + 5*e  x2  4.75  0.75

>> delete plusmult, s:
```

Changes:

- `stats::tabulate` is a new function.

`stats::unzipCol` – extract columns from a list of lists

`stats::unzipCol(list)` extracts the columns of a matrix structure encoded by a list of lists.

Call(s):

- `stats::unzipCol(list)`

Parameters:

`list` — a list of lists.

Return Value: a sequence of lists to be regarded as columns.

Related Functions: `stats::col`, `stats::sample2list`, `stats::zipCol`

Details:

- `stats::unzipCol` treats a list of lists like a list of rows of a `stats::sample` and extracts the columns. In conjunction with `stats::sample2list` it is useful for extracting the columns of a `stats::sample`.
- `stats::unzipCol` is the inverse of `stats::zipCol`.

Example 1. We extract the columns from a list of rows representing a matrix structure:

```
>> stats::unzipCol([[a11, a12], [a21, a22], [a31, a32]])

                [a11, a21, a31], [a12, a22, a32]
```

Example 2. A list of rows is used to create a sample:

```
>> stats::sample([[123, s, 1/2], [442, s, -1/2], [322, p, -
1/2]])

                          123  s   1/2
                          442  s  -1/2
                          322  p  -1/2
```

We re-convert the sample to a list of lists:

```
>> stats::sample2list(%)

         [[123, s, 1/2], [442, s, -1/2], [322, p, -1/2]]
```

Finally, we extract the columns:

```
>> stats::unzipCol(%)

          [123, 442, 322], [s, s, p], [1/2, -1/2, -1/2]
```

Changes:

- `stats::unzipCol` is a new function.

`stats::variance` – the variance

`stats::variance(data)` returns the variance of the data.

Call(s):

- `stats::variance(x1, x2, ... <, Sample>)`
- `stats::variance([x1, x2, ...] <, Sample>)`
- `stats::variance(s <, c> <, Sample>)`

Parameters:

`x1, x2, ...`	— the statistical data: arithmetical expressions.
`s`	— a sample of domain type `stats::sample`.
`c`	— an integer representing a column index of the sample `s`. This column provides the data `x1`, `x2` etc.

Options:

`Sample`	— with this option the given data are regarded as a "sample", not a as a full population.

Return Value: an arithmetical expression.

Related Functions: `stats::a_quantil`, `stats::geometric`, `stats::harmonic`, `stats::mean`, `stats::median`, `stats::modal`, `stats::quadratic`, `stats::stdev`

Details:

- The variance of n values $x_1, \ldots, x_n$ is $\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$, where $\bar{x}$ is the arithmetic mean of the x_i. With the option *`Sample`* the value $\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$ is returned instead.
- The column index `c` is optional, if the data are given by a `stats::sample` object containing only one non-string column. Cf. example 3.

Example 1. We calculate the variance of three values:

```
>> stats::variance(2, 3, 5)
                                14/9
```

Alternatively, the data may be passed as a list:

```
>> stats::variance([2, 3, 5])
                                14/9
```

Example 2. We create a sample:

```
>> stats::sample([[a1, b1, c1], [a2, b2, c2]])
```

```
                          a1  b1  c1
                          a2  b2  c2
```

The variance of the second column is:

```
>> expand(stats::variance(%, 2))
```

```
                           2             2
                         b1    b1 b2   b2
                         --- - ----- + ---
                          4      2      4
```

Example 3. We create a sample consisting of one string column and one non-string column:

```
>> stats::sample([["1996", 1242], ["1997", 1353], ["1998", 1142]])
```

```
                          "1996"  1242
                          "1997"  1353
                          "1998"  1142
```

We compute the variance of the second column. In this case this column does not have to be specified, since it is the only non-string column:

```
>> float(stats::variance(%))
```

```
                            7426.888889
```

We repeat the computation with the option *Sample*:

```
>> float(stats::variance(%2, Sample))
```

```
                            11140.33333
```

Changes:

- The input data can now be provided by a `stats::sample`.

`stats::zipCol` – convert a sequence of columns to a list of lists

`stats::zipCol(..)` converts a sequence of columns to a format suitable for creating a `stats::sample`.

Call(s):

- `stats::zipCol(column1, column2, ..)`

Parameters:

`column1, column2, ..` — lists.

Return Value: a list of lists.

Related Functions: `stats::sample2list`, `stats::unzipCol`

Details:

- `stats::zipCol` is useful for converting column data given in lists to a list of lists accepted by `stats::sample`.
- `stats::zipCol` is the inverse of `stats::unzipCol`.

Example 1. We convert a single column to a nested list:

```
>> stats::zipCol([a, b, c])
                          [[a], [b], [c]]
```

This list is accepted by `stats::sample`:

```
>> stats::sample(%)
                                a
                                b
                                c
```

Example 2. We build a sample consisting of two columns:

```
>> column1 := [122, 442, 322]: column2 := [s, s, p]:
   stats::zipCol(column1, column2)
                 [[122, s], [442, s], [322, p]]
>> stats::sample(%)
                             122  s
                             442  s
                             322  p
```

Changes:

- `stats::zipCol` is a new function.