Matrix creation:

' ' (space) and ',' (comma) separate columns; ';' (semicolon) separates rows

    m23 = [1 2 3; 4 3 1] - 2 rows, 3 columns   m6x6 = [m23 m23; m23 m23; m23 m23]

`zeros`, `ones` – these functions create matrices filled with just one value

    `zeros(3)` – square 3x3 matrix   `zeros(4,3)` – 4 rows, 3 columns

`eye` – creates matrix „one" (ones on the main diagonal, zeros everywhere else)

`repmat(m, r, c)` – replication of matrix `m`: in each superrow there are `c` copies of `m`, in supercolumn there are w `r` copies of `m`   `repmat(m23, 3, 2)` – 6 rows, 6 columns

Sequences:

`start_val:end_val` – in this form the default value of step is used (= 1)

    1:5 -> [1 2 3 4 5]

`start_val:step:end_val` – 3:-0.5:1 -> [3.0 2.5 2.0 1.5 1.0]

Operators:

„Ordinary" operators perform matrix operations. In case of addition and subtraction it doesn't really matter, but it is very important in multiplication (`*`) and division (`/`) – you have to pay attention to dimensions of both arguments; exponentiation (`^`) is applicable only to square matrices.
All operators have „element by element" versions: addition (`.+`) and subtraction (`.−`) operates in the same way; multiplication (`.*`) and division (`./`) is performed on pairs of elements from both arguments – matrices need to have the same size; in exponentiation (`.^`) each matrix element is exponentiated independently.

Very important (and often used) is unary transposition operator ''' (apostrophe)

Comparison operators operate in „element by element" manner producing logical values (the result has the same size as both arguments).

Indexing:

We use notation: `m(rows, columns)` while indexing, where `rows` and `columns` are sets of indices of rows and columns we want to select; indexing starts with „1"!
Scalar: `m(1,1)` – selecting an element from the first row and the first column („upper-left")
Enumeration: `m([1 5], 1)` – selection of elements from rows 1 and 5 in the first column
Sequence: `m(1:3, 2)` – selection of elements from rows 1-3 in the second column
end: `m(4, 2:end)` – selection of elements in the 4th row from 2nd to the last column
All range: `m(:, 2:4)` – selection of elements in all rows in columns from 2nd to 4th
Supplying just one index value produces vector result: `eye(3)(:) -> [1 0 0 0 1 0 0 0 1]'`

Logical values can be used as indices, under conditio that their number is lower than or equal to the number of matrix elements in given dimension.
`m6x6(1:6 != i, :)` – selects from matrix `m6x6` all rows without ith row.

Aggregation:

Aggregation functions have usually unary and binary versions (i.e. with just one or with two arguments). In the first case default „direction" of computation is used: having constant value of column index aggregating function iterates over all rows. In the second case we explicitly impose iteration over rows (DIM = 1) or columns (DIM = 2).

This explicitness is vital when an aggregation argument may reduce to vector.

`sum` – sum of element: `sum(1:5) -> 15`   `sum(1:5,1) -> [1 2 3 4 5]`

`sumsq` – sum of elements squared   `sumsq(1:5) -> 55`

`prod` – product of element   `prod(1:5) -> 120`

`mean` – mean value (average)   `mean(1:5) -> 3`   `mean(1:5, 1) -> [1 2 3 4 5]`

`std` – standard deviation; **beware**: the second argument is the normalization method selector!

   `std(1:5) -> 1.5811`   `std(1:5, 1) -> 1.4142`   `std(1:5,[],1) -> [0 0 0 0 0]`

`var` – variation; **beware**: the second argument is the normalization method selector!

`min` – minimum value; **beware**: the second argument is the value to compare with!

   `min(1:5) -> 1`   `min(1:5, 1) -> [1 1 1 1 1]`   `min(1:5,[],1) -> [1 2 3 4 5]`

`max` – maximum value; **beware**: the second argument is the value to compare with!

Version with two output arguments:

`[min_v, min_id] = min([5:-1:0 1:5])`   `->`   `min_v -> 0 min_id -> 6`