Programming guide

Preface

This interface: *standard interface*

Document/Worksheet mode, *1-D/2-D* math notation

http://www.maplesoft.com/
Maplesoft Application Center / MaplePrimes / Student Resource Center

1. Introduction

Maple = Kernel (built-in commands, kernel extensions) + Math library

statement; - computed and displayed
statement: - computed but not displayed

?command - get help for command

Quick help: **F1**

Help on command at cursor position: **F2**

Switch between text/math by **F5**

Delete input/output: **CTRL + Del**

String:

```
> "Hello World";
```

"Hello World" **(1.2.1)**

Evaluate expression:

```
> a := 5/3;
```

$$a := \frac{5}{3} \quad \textbf{(1.2.2)}$$

```
> evalf(a);
```

1.666666667 **(1.2.3)**

Procedure:

```
> GetAngle := proc( opposite, hypotenuse )
          local theta;
          theta := arcsin(opposite/hypotenuse);
          evalf(180/Pi*theta);
  end proc;
```

GetAngle := **proc**(*opposite*, *hypotenuse*) **(1.2.4)**

  local θ;

  $\theta := \arcsin(opposite / hypotenuse);\ evalf(180 * \theta / \pi)$

end proc

```
> GetAngle(4,5);
```

53.13010234 **(1.2.5)**

Use **Shift** + **Enter** to enter multiline expressions

```
> # single line comment
> (* Multiline
  comment*)
```

Name evaluation:

```
> restart;
```

```
b:=a; a:=1; b; a:=2; b;
```

$$b := a$$
$$a := 1$$
$$1$$
$$a := 2$$
$$2 \quad \textbf{(1.2.6)}$$

```
> restart;
  a:=1; b:=a; b; a:=2; b;
```

$$a := 1$$
$$b := 1$$
$$1$$
$$a := 2$$
$$1 \quad \textbf{(1.2.7)}$$

Clear memory and unbind all variable names

```
> restart;
```

Don't forget:

- end statements with ";" or ":"
- multiplication only works with "*"

Equation:

```
> x^2-2*x+1=10;
```

$$x^2 - 2x + 1 = 10 \quad \textbf{(1.2.8)}$$

```
> solve(%, x);
```

$$1 + \sqrt{10}, 1 - \sqrt{10} \quad \textbf{(1.2.9)}$$

"%", "%%" and "%%%" reference previous results

2. Maple Language Elements

Reserved keywords: **break**, **next, if**, **then**, **elif**, **else**, **for**, **from**, **in**, **by**, **to**, **while**, **do**, **proc**, **local**, **global**, **option**, **error**, **return**, **options**, **description**, **export**, **module**, **use**, **end**, **assuming**, **try**, **catch**, **finally**, **read**, **save**, **quit**, **done**, **stop**, **union**, **minus**, **intersect**, **subset**, **and**, **or**, **not**, **xor**, **implies**, **mod**

Operators:

! factorial
$ creating sequence
@ function composition
, expression sequence separator
|| string/name concatenation
. matrix multiplication
-> defining function
.. range
mod modulo
<> not equal
union, intersect, minus, subset, in set theory
and, or, xor, implies, not logic
Op~ apply Op elementwise
%, %%, %%% reference previous results (ditto operators)

Names:

```
> My_name_1, `Ez is egy név` := 1, 2;
```

(1.3.1)

$My\_name\_1, Ez\ is\ egy\ név := 1, 2$ **(1.3.1)**

```
> `Ez is egy név`;
```

2 **(1.3.2)**

Names are case-sensitive.

Type:

```
> type(My_name_1, 'integer');
  type(`Ez is egy név`, 'string');
```

true

false **(1.3.3)**

Strings:

```
> S:="Hello world";
```

S := "Hello world" **(1.3.4)**

```
> length(S);
```

11 **(1.3.5)**

```
> S[1], S[6], S[11]; # indexing
```

"H", " ", "d" **(1.3.6)**

```
> S[6..9], S[-6..-1], S[-2..11], S[5..]; # slices
```

" wor", " world", "ld", "o world" **(1.3.7)**

```
> SearchText("my s", "This is my string."); # case sensitive
  search
```

9 **(1.3.8)**

```
> searchtext("My S", "This is my string."); # case insensitive
```

9 **(1.3.9)**

```
> SearchText("is", "This is my string.", 4..-1); # constrained
  search
```

3 **(1.3.10)**

```
> i := 5; cat("The value of i is ", i, ".");
```

$i := 5$

"The value of i is 5." **(1.3.11)**

```
> filename := cat(kernelopts(mapledir), kernelopts(dirsep),
  "lib");
```

filename := "C:\Program Files\Maple 17\lib" **(1.3.12)**

Escaping special characters: "\\", "\"", ...

Parse string to expression:

```
> parse("a+b/c");
  parse("sin(Pi)");
  parse("sin(Pi)", 'statement'); # with the 'statement' option,
  the result is evaluated
```

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

$$\sin(\pi)$$

$$0$$ **(1.3.13)**

Convert expression to string:

```
> convert(a, 'string');
  convert(42, 'string');
```

"a"

(1.3.14)

"42" **(1.3.14)**

Multiline tokens:

```
> 958477383\
  24324;
```

95847738324324 **(1.3.15)**

```
> "this is a \
  long string";
```

"this is a long string" **(1.3.16)**

```
> "this is a "
  "long string";
```

"this is a long string" **(1.3.17)**

Suppress evaluation with single quotes:

```
> sin(0);
  'sin'(0);
```

0

sin(0) **(1.3.18)**

This can be used to pass names to functions, even if there is an assigned a value

Brackets:

() group expressions
[] indexing, lists
{} sets
<> vectors and matrices

Passing command to operating system:

!<*command*>;

Type and operands of expression:

```
> op(0, a+1);
  op(1, a+1), op(0, op(1, a+1));
  op(2, a+1), op(0, op(2, a+1));
```

`+`

a, *symbol*

1, *Integer* **(1.3.19)**

```
> op(0, f(x));
  op(1, f(x));
```

f

x **(1.3.20)**

Maple integer subtypes: *integer[8], integer[4], negint, posint, nonnegint, nonposint, even, odd, prime*

Structured types:

```
> type([-1, 2, 11], 'list({negint,prime})');
  type([0, 2, 11], 'list({negint,prime})');
```

true

false **(1.3.21)**

Expression DAG:

```
> eq := x=y+x:
  dismantle(eq);
```

```
EQUATION(3)
   NAME(4): x
```

```
SUM(5)
    NAME(4): y
    INTPOS(2): 1
    NAME(4): x
    INTPOS(2): 1
```

3. Maple Expressions

Maple applies automatic simplification and evaluation to expressions:

```
> a:=3;
  x*(a+x/x);
```

$$a := 3$$

$$4\,x \quad (1.4.1)$$

A name evaluates to the last object of the evaluation chain by default. If the name is enclosed in unevaluation quotes, then it evaluates to itself.

```
> unassign('a', 'b');
  a:=b; b:=1;
  a;
  'a';
```

$$a := b$$

$$b := 1$$

$$1$$

$$a \quad (1.4.2)$$

Constructors:

```
> `+`(3, 4, 5);
```

$$12 \quad (1.4.3)$$

Names beginning with an underscore character (_) are reserved for use by the Maple library. Names beginning with "_Env" are treated as environment variables.

Maple has predefined constants:

```
> constants;
```

$$\mathit{false}, \gamma, \infty, \mathit{true}, \mathit{Catalan}, \mathit{FAIL}, \pi \quad (1.4.4)$$

Maple also has several other special constants, such as **NULL, undefined, I, Digits, ...** **NULL** stands for the empty sequence, **I** is the imaginary unit.

Other protected names: *sin, diff, degree, int, list, ...*

Protecting names and options by unevaluation:

```
> output:=13;
  CodeGeneration:-C( x^2, 'output' = 'string' ); # 'output'
  evaluates to an option name, and not to the previously
  assigned value 13.
```

$$\mathit{output} := 13$$

"cg0 = x * x;
" (1.4.5)

Summation on generic index:

```
> B := <1,2,3,4>;
  sum('B[i]', i = 1..4);
```

$$B := \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

$$10 \qquad \textbf{(1.4.6)}$$

Dividing with remainder:

```
> remainder := irem(45,3,'quotient');
  quotient;
```

$$remainder := 0$$

$$15 \qquad \textbf{(1.4.7)}$$

Unassigning names (both work):

```
> a := 'a';
```

$$a := a \qquad \textbf{(1.4.8)}$$

```
> unassign(a);
```

Treating a function as an unevaluated expression:

```
> 'sin'(pi);
```

$$\sin(\pi) \qquad \textbf{(1.4.9)}$$

Defining a procedure, that returns unevaluated whenever it encounters unprocessable arguments:

```
> f := proc(x)
    if type(x, 'numeric') then
      if x > 0 then
        x
      else
        2
      end if
    else
      'procname(_passed)'
    end if
  end proc:
```

Maximum number of digits:

```
> kernelopts('maxdigits');
```

$$38654705646 \qquad \textbf{(1.4.10)}$$

Number of digits in an integer:

```
> length(421324364635);
```

$$12 \qquad \textbf{(1.4.11)}$$

Constructing fractions:

```
> -2/3 = Fraction(-2, 3);
```

$$-\frac{2}{3} = -\frac{2}{3} \qquad \textbf{(1.4.12)}$$

Numerator and denominator:

```
> numer(-2/3);
  denom(-2/3);
```

$$-2$$

$$3 \qquad \textbf{(1.4.13)}$$

Exponent form of floats:

```
> 2.14E10;
```

```
2.14E-9;
.00023E5;
2.E3; #invalid!
```

$$2.14\ 10^{10}$$

$$2.14\ 10^{-9}$$

$$23.$$

Error, missing operator or `;`

Minimum and maximum values of the exponent:

```
> Maple_floats('MIN_EXP');
  Maple_floats('MAX_EXP');
```

$$-9223372036854775806$$

$$9223372036854775806 \quad \textbf{(1.4.14)}$$

Creating software floats:

```
> SFloat(23, -1);
```

$$2.3 \quad \textbf{(1.4.15)}$$

Extracting significand (mantissa) and exponent from a float:

```
> SFloatMantissa(92.35345);
  SFloatExponent(92.35345);
```

$$9235345$$

$$-5 \quad \textbf{(1.4.16)}$$

```
> op(92.35345);
```

$$9235345, -5 \quad \textbf{(1.4.17)}$$

Two software floats are equal if they represent the same number:

```
> evalb( 2.3 = 2.30 );
  op(2.3);
  op(2.30);
```

$$true$$

$$23, -1$$

$$230, -2 \quad \textbf{(1.4.18)}$$

However,

```
> evalb(<2.3,4.5> = <2.30,4.50>);
  EqualEntries(<2.3,4.5>, <2.30,4.50>);
```

$$false$$

$$true \quad \textbf{(1.4.19)}$$

Hardware floats have low-level implementation and are binary-based, therefore, conversion to them often results in rounding errors.

```
> HFloat(24375, -3);
```

$$24.3750000000000 \quad \textbf{(1.4.20)}$$

```
> op( HFloat(2.3) );
```

$$229999999999999982, -17 \quad \textbf{(1.4.21)}$$

Complex numbers:

```
> I^2;
  2 - 3*I;
  a + b*I;
```

$$-1$$

$$2 - 3\,I$$

$$a + I\,b \quad \textbf{(1.4.22)}$$

Complex number constructor:
- 1-argument form:

```
> Complex(2), Complex(0), Complex(0.5);
```

$$2\,I, 0, 0.5\,I \quad (1.4.23)$$

```
> Complex(2 - 3*I), Complex(infinity), Complex(undefined);
```

$$2 - 3\,I, \infty\,I, \mathit{undefined}\,I \quad (1.4.24)$$

- 2-argument form:

```
> Complex(2, -3), Complex(2.1, 3), Complex(0, 0);
```

$$2 - 3\,I, 2.1 + 3.\,I, 0 \quad (1.4.25)$$

All forms of complex infinity are numerically equivalent and they are all treated as distinct from real infinities:

```
> evalb(infinity + infinity*I = infinity - infinity*I);
```

$$\mathit{true} \quad (1.4.26)$$

```
> evalb(-infinity + infinity*I = -infinity);
```

$$\mathit{false} \quad (1.4.27)$$

Real and imaginary parts:

```
> Re(2.3 + sqrt(2)*I);
  Im(2.3 + sqrt(2)*I);
```

$$2.3$$

$$\sqrt{2} \quad (1.4.28)$$

In the expression $a + b \cdot I$, a and b are not assumed to be real. To make this assumption, we can use the *evalc* command.

```
> Re(a + b*I);
```

$$\Re(a + I\,b) \quad (1.4.29)$$

```
> evalc(Re(a + b*I));
```

$$a \quad (1.4.30)$$

Indexing expressions: *expr[index]*, where *expr* is any expression and *index* is a sequence of expressions.

```
> 2[3, 4];
  a[];
  a[1];
  a[b, c];
  map[a];
  [1, 2, 3][2..3][1];
```

$$2_{3,\,4}$$

$$a_{[\,]}$$

$$a_1$$

$$a_{b,\,c}$$

$$\mathit{map}_a$$

$$2 \quad (1.4.31)$$

With constructor:

```
> `?[]`( S, [ a, b, c ] );
```

$$S_{a,\,b,\,c} \quad (1.4.32)$$

Operands are the indices:

```
> nops(a[b, c, d]);
```

```
op(a[b, c, d]);
op(2, a[b, c, d]);
op(2..3, a[b, c, d]);
```

$$3$$
$$b, c, d$$
$$c$$
$$c, d \tag{1.4.33}$$

Selection operation: if *expr* is a sequence, the index sequence must evaluate to a positive integer, an integral range, a list of integers or the empty sequence. Negative indices select elements starting from the end of the list. Left or right bound of the integral range can be omitted.

```
> expr := (1,2,3,4,5);
  expr[1];
  expr[1..2];
  expr[..-3];
  expr[[4, 3, 2, 1]];
  expr[];
```

$$expr := 1, 2, 3, 4, 5$$
$$1$$
$$1, 2$$
$$1, 2, 3$$
$$4, 3, 2, 1$$
$$1, 2, 3, 4, 5 \tag{1.4.34}$$

```
> # invalid selection:
  expr[0];
  expr[88];
  expr[1, 2, 3];
  expr[-2..1];
Error, invalid subscript selector
Error, invalid subscript selector
Error, invalid subscript selector
Error, invalid subscript selector
```

Selection works similarly for sets, lists, vectors, arrays and matrices. Matrices and multidimensional arrays use multiple index sequences.

```
> S := {2, 3, 5, 7, 11};
  L := [100, 90, 80, 70, 60];
  V := <12, 14, 16, 18, 20>;
  A := Array([-9, -10, -11, -12, -13]);
  A2 := Array(2..3,-1..1,[[1, 2, 3], [4, 5, 6]]); # first 2
  arguments are the index ranges (they don't start from 1)
  M := Matrix(3, 2, [[21, 22], [23, 24], [25, 26]]); # or: M :=
  <21, 22; 23, 24; 25, 26>;
```

$$S := \{2, 3, 5, 7, 11\}$$
$$L := [100, 90, 80, 70, 60]$$
$$V := \begin{bmatrix} 12 \\ 14 \\ 16 \\ 18 \\ 20 \end{bmatrix}$$

$$A := \begin{bmatrix} -9 & -10 & -11 & -12 & -13 \end{bmatrix}$$

$A2 := Array(2..3, -1..1, \{(2, -1) = 1, (2, 0) = 2, (2, 1) = 3, (3, -1) = 4, (3, 0) = 5, (3, 1) = 6\}, datatype = anything, storage = rectangular, order = Fortran\_order)$

$$M := \begin{bmatrix} 21 & 22 \\ 23 & 24 \\ 25 & 26 \end{bmatrix} \quad (1.4.35)$$

```
> S[4]; # relying on the internal ordering of a set should be
  avoided, as it may be different from the one given in the
  set's definition
  L[4];
  L[2..-2];
  V[-2];
  A[4];
  A(4); # programming-style indexing
```

$$7$$
$$70$$
$$[90, 80, 70]$$
$$18$$
$$-12$$
$$-12 \quad (1.4.36)$$

```
> A2[1, 1]; # invalid, indices are out of the given range
  A2[3, 0];
  A2(1, 1); # relative indexing
  A2[2..3, [-1, 1]];
  M[2, 2];
  M[[1, 3], 1]; # submatrix selection
  M(3, 1); # programming notation
  M(4); # fourth element, matrix treated as a vector by
  concatenating columns
```

Error, Array index out of range

$$5$$
$$1$$

$Array(2..3, 1..2, \{(2, 1) = 1, (2, 2) = 3, (3, 1) = 4, (3, 2) = 6\}, datatype = anything, storage = rectangular, order = Fortran\_order)$

$$24$$
$$\begin{bmatrix} 21 \\ 25 \end{bmatrix}$$
$$25$$
$$22 \quad (1.4.37)$$

Arrays, vectors and matrices are collectively known as *rtable*s. Programming-style indexing can only be used with rtables.

Indexing a name results in an indexed name. Indexed names can be assigned to a value.

```
> aName[x^2 - 3*x, "a string", anotherName[2, b]] := 2;
  aName[x^2 - 3*x, "a string", anotherName[2, b]];
```

$$aName_{x^2 - 3x, \text{"a string"}, anotherName_{2, b}} := 2$$

$$2 \qquad \textbf{(1.4.38)}$$

Tables are key-value stores:

```
> T := table([a = 1, b = 2, (c,d) = 3]);
```

$$T := table([b = 2, a = 1, (c, d) = 3]) \qquad \textbf{(1.4.39)}$$

```
> T[a], T[(c, d)];
```

$$1, 3 \qquad \textbf{(1.4.40)}$$

If the value is not found for a key, an indexed expression will be returned.

```
> T[0], T[d], T[u, v], T[];
```

$$2, T_d, T_{u, v}, T_{[\,]} \qquad \textbf{(1.4.41)}$$

```
> T[0] := 2;
  T[0];
```

$$T_0 := 2$$

$$2 \qquad \textbf{(1.4.42)}$$

Strings can be indexed, too. If the index is an integer or an integral range, a substring will be returned. Otherwise, the indexed string expression is returned but is can't be used as a variable name.

```
> "abcde"[3];
  "abcde"[2..4];
  "abcde"[u, v^2 - s*t];
  "abcde"[];
  "abcde"[[2, 5]];
```

"c"

"bcd"

$$\text{"abcde"}_{u, v^2 - s t}$$

$$\text{"abcde"}_{[\,]}$$

$$\text{"abcde"}_{[2, 5]} \qquad \textbf{(1.4.43)}$$

```
> u := 5;
  "abcde"[u, v^2 - s*t];
  "abcde"[u, v^2 - s*t] := 4;
```

$$u := 5$$

$$\text{"abcde"}_{5, v^2 - s t}$$

Error, invalid left hand side of table reference

Modules can also be indexed by the name of an export.

```
> m := module() export e, f:= 2; end module:
> m[f];
```

$$2 \qquad \textbf{(1.4.44)}$$

The same effect can be achieved by the member selection operator. The difference is that the index selection form will evaluate f before resolving the export.

```
> m:-f;
  f := 3;
  m[f];
```

$$2$$

$$f := 3$$

```
Error, index must evaluate to a name when indexing a module
```

```
> evalb(m:-e = e);
```

$$false \quad (1.4.45)$$

The 1-argument form of the member selection operator can be used to reference a global name (for example, inside a procedure).

```
> t := 10;
  pp := proc()
    local t;
    print(t);
    print(:-t);
  end proc:
  pp();
```

$$t := 10$$
$$t$$
$$10 \quad (1.4.46)$$

Function call (function expression):

```
> F();
  F(x);
  F(x + y);
  F(x, y);
  sin(x + y);
```

$$F(\ )$$
$$F(x)$$
$$F(x+y)$$
$$F(x, y)$$
$$\sin(x+y) \quad (1.4.47)$$

Operands of function call:

```
> op(F(x, y, z));
  nops(F(x, y, z));
  nops(F());
  op(0, F(x, y, z));
```

$$x, y, z$$
$$3$$
$$0$$
$$F \quad (1.4.48)$$

Operator algebra:
- @ is the composition operator
- numeric quantities can be used as constant functions

```
> (f^2 + g@h - 2)(x);
  2(x);
  2(x, y, z);
```

$$f(x)^2 + g(h(x)) - 2$$
$$2$$
$$2 \quad (1.4.49)$$

The function itself can be any expression that evaluates to a procedure.

Arithmetic operators in Maple: +, -, *, /, ^

Polynomials:

```
> 2*x^2 - 5*x + 12;
```

$$2\,x^2 - 5\,x + 12 \quad (1.4.50)$$

```
> op(u + v);
  op(0, u + v);
  nops(u + v);
```

$$u, v$$
$$`+`$$
$$2 \quad (1.4.51)$$

```
> op(u - v);
  op(0, u - v);
  nops(u - v);
```

$$u, -v$$
$$`+`$$
$$2 \quad (1.4.52)$$

```
> type(u - v, '`+`');
```

$$true \quad (1.4.53)$$

Addition is a multi-operand operator in Maple:

```
> nops(a + b + c + d + e);
```

$$5 \quad (1.4.54)$$

Constructor for sum:

```
> `+`(a, b, c);
```

$$a + b + c \quad (1.4.55)$$

Maple performs automatic simplification on sums:

```
> a + 2 + b + 3 + c + 4;
  '2/3 + sin(5*Pi/6 - 2*Pi/3)';
  2/3 + sin(5*Pi/6 - 2*Pi/3);
```

$$a + 9 + b + c$$
$$\frac{2}{3} + \sin\left(\frac{1}{6}\,\pi\right)$$
$$\frac{7}{6} \quad (1.4.56)$$

If any operands of a sum is a float, the result is computed as a float.

Addition with *infinity* and *undefined*:

```
> -0.0;
  2.3 + infinity;
  infinity - infinity;
  2.3 + undefined;
```

$$-0.$$
$$Float(\infty)$$
$$undefined$$
$$Float(undefined) \quad (1.4.57)$$

Sum of lists, vectors and matrices of the same length:

```
> [a, b, c] + [x, y, z];
  <1, 2; 3, 4> + <5, 6; 7, 8>;
```

$$[x + a, y + b, z + c]$$

(1.4.58)

$$\begin{bmatrix} 6 & 8 \\ 10 & 12 \end{bmatrix} \quad (1.4.58)$$

Lists, vectors and matrices of different length cannot be summed:

```
> [1, 2] + [1, 2, 3];
Error, adding lists of different length
```

Multiplication/division vorks the same way as addition/substraction.

```
> nops(6 * x * y);
  op(6 * x * y);
  op(0, 6 * x * y);
```

$$3$$
$$6, x, y$$
$$`*` \quad (1.4.59)$$

```
> type(a*b, '`*`');
  type(a/b, '`*`');
```

true

true **(1.4.60)**

The *dismantle* command prints the internal representation of the expression. $\frac{a}{b}$ is represented as $a \cdot b^{-1}$:

```
> dismantle(a/b);

PROD(5)
   NAME(4): a
   INTPOS(2): 1
   NAME(4): b
   INTNEG(2): -1
```

Automatic simplification occurs:

```
> '2 * 3 * x * y';
  op(2 * 3 * x * y);
```

$$6\,x\,y$$
$$6, x, y \quad (1.4.61)$$

Constructor for products:

```
> `*`(a, b, c);
```

$$a\,b\,c \quad (1.4.62)$$

As with addition, if any operands of a product is a float, the result is computed as a float. This, however, does not extend into function calls:

```
> '2.3*(5*Pi/6 - 2*Pi/3)';
  '2.3*sin(5*Pi/6 - 2*Pi/3)';
```

$$0.3833333333\,\pi$$
$$2.3\,\sin\left(\frac{1}{6}\,\pi\right) \quad (1.4.63)$$

Lists, arrays and vectors can be multiplied by a number, and two arrays of the same length can be multiplied elementwise. For elementwise multiplication of lists and vectors, use the `*~` operator.

```
> 2*[1, 2, 3];
```

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

$$[[2, 4, 6]]$$

$$\begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 4 & 6 \end{bmatrix} \quad \textbf{(1.4.64)}$$

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

$$[1, 2, 3]^2$$

Error, (in rtable/Product) invalid arguments

$$\begin{bmatrix} 1 & 4 & 9 \end{bmatrix} \quad \textbf{(1.4.65)}$$

```
> [1, 2, 3]*~[1, 2, 3];
  <1, 2, 3>*~<1, 2, 3>;
  Array([1, 2, 3])*~Array([1, 2, 3]);
```

$$[[1, 4, 9]]$$

$$\begin{bmatrix} 1 \\ 4 \\ 9 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 4 & 9 \end{bmatrix} \quad \textbf{(1.4.66)}$$

The same rules apply to division:

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

$$\left[\left[\frac{1}{2}, 1, \frac{3}{2}\right]\right]$$

$$\begin{bmatrix} \frac{1}{2} \\ 1 \\ \frac{3}{2} \end{bmatrix}$$

$$\begin{bmatrix} \frac{1}{2} & 1 & \frac{3}{2} \end{bmatrix} \quad \textbf{(1.4.67)}$$

```
> [1, 2, 3]/~[1, 2, 3];
  <1, 2, 3>/~<1, 2, 3>;
  Array([1, 2, 3])/~Array([1, 2, 3]);
```

$$[[1, 1, 1]]$$

$$\begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \quad \textbf{(1.4.68)}$$

For non-commutative matrix product, use the `.` (dot) operator (see later).

Powers are formed by using the `^` operator.

```
> a^b;
```

$$a^b \quad \textbf{(1.4.69)}$$

Nested exponentiation must be written with parenthesis:

```
> a^(b^c);
  (a^b)^c;
  a^b^c;
```

$$a^{b^c}$$

$$\left(a^b\right)^c$$

Error, ambiguous use of `^`, please use parentheses

Rational powers represent roots. Roots of floating point numbers are computed during automatic simplification.

```
> 5^(1/2);
  2^(3/2);
  2^(2/3);
  1.524^(1/3);
```

$$\sqrt{5}$$

$$2\sqrt{2}$$

$$2^{2/3}$$

$$1.150787111 \quad \textbf{(1.4.70)}$$

If the exponent is a float, the result will be a float as well (or sometimes even a complex number)
:

```
> 1^1.1;
  (-10)^1.0;
  (-10)^0.0;
```

$$1.$$

$$-10.$$

$$1. + 0.\,I \quad \textbf{(1.4.71)}$$

Some undefined forms:

```
> 0^0;
  0^0.0;
  a^0;
  a^0.0;
  infinity^0;
  infinity^0.0;
  undefined^0;
  undefined^5;
  5^undefined;
  infinity^(-infinity);
  (-infinity)^(infinity);
```

$$1$$
$$Float(undefined)$$
$$1$$
$$1.0$$
$$1$$
$$Float(undefined)$$
$$1$$
$$undefined$$
$$undefined$$
$$0$$
$$\infty + \infty\, I \tag{1.4.72}$$

Powers of arrays are done elementwise but powers of matrices are computed with matrix product.

```
> Array([[1, 2], [3, 4]])^3;
  <1, 2; 3, 4>^3;
```

$$\begin{bmatrix} 1 & 8 \\ 27 & 64 \end{bmatrix}$$
$$\begin{bmatrix} 37 & 54 \\ 81 & 118 \end{bmatrix} \tag{1.4.73}$$