LINE 3.0 Manual

Language with Intuitive and Natural Expression

Contents

1 Introduction

1.1 What is LINE?

LINE is a programming language designed to be understandable and easy to learn. The syntax puts logic at its centre, making it readable almost like natural text.

1.2 What does LINE stand for?

LINE is an acronym: **Language with Intuitive and Natural Expression**.

A LINE program is saved in a file with the `.line` extension and run by the interpreter.

2 Comments

```
>> Single-line comment

<* Multi-line
   comment *>
```

3 Escape

The escape character is the backslash \. It precedes special characters to give them a different meaning from their literal one.

Sequence	Meaning
`\n`	New line (newline)
`\\`	Literal backslash
`\@`	Literal `@` (not a variable reference)
`\%`	Literal `%` (`%` without escape is invisible in `TALK`)
`\;`	Literal `;` in a list (not an element separator)
`\\|`	Literal `|` in a list (not a tag separator)
`\tNAME`	Type of variable `NAME` (returns `string`, `number` or `bool`)

Note on %: the `%` symbol written directly inside a `TALK` is invisible in the output. If stored in a variable, it is printed normally. To print it directly use `\%`.

4 Variables

A variable has this form:

```
VAR_TYPE var_name = data
```

- `VAR_TYPE` — indicates the type of variable (see section 6). It can be omitted for standard variables.
- `var_name` — unique name. It can contain only letters, digits (not in the first position) and the underscore `_`.

- `data` — the stored value. If there is more than one piece of data it is a list (see section 15).

To refer to the value of a variable from another point in the code, use `@` before the name:

```
name = Mario
TALK Hello, @name!
<* Output: Hello, Mario! *>
```

The "reach" of `@` stops at the first character not allowed in a name (space, `!`, `,`, etc.).

5 Data types

There are three data types:

Type	Prefix	Description
String	*(none)*	Stores the data as-is, as text
Numeral	`#`	Stores a numeric value
Boolean	`#`	Only two values: `V` (true) or `F` (false)

The prefix `#` before the name declares that the variable contains a numeral or a boolean:

```
var = 8      >> string: stores the text "8"
#var = 8     >> numeral: stores the number 8
var = V      >> string: stores the text "V"
#var = V     >> boolean: stores the true value
```

If something that cannot be either a number or a boolean is stored in a variable with `#`, a type error is raised.

5.1 Type conversions

Command	Effect
`n:var`	Converts to numeral. If the string is a number that value is used, otherwise it equals `1`. Empty string → `0`.
`b:var`	Converts to boolean. Empty string or `"F"` → `F`, everything else → `V`. From a number: `0` → `F`, any other → `V`.
`s:var`	Converts to string. The value becomes readable text.

```
var = 8
n:var        >> now var is the numeral 8

#n = 0
b:n          >> now n is the boolean F

#n = 5
s:n          >> now n is the string "5"
```

To find out the current type of a variable, use `\t` followed by the name:

```
#age = 25
TALK \tage   >> Output: number
```

```
name = Mario
TALK \tname  >> Output: string
```

6 Variable types

There are three types of variable.

6.1 Standard *(no prefix)*

The most common variable. The value can be changed with a **reassignment** — same syntax as the declaration:

```
var = Good morning   >> declaration
var = Goodbye        >> reassignment
```

The type can also be changed during reassignment:

```
var = Hello    >> string
#var = V       >> reassigned as boolean
```

> **Important:** if a variable is set equal to another and that one changes, the first **will not change** — the value is copied at the moment of assignment.

```
#a = 5
var = @a    >> var is 5 (copy of a)
#a = 6      >> a changes, but var remains 5
```

6.2 STAY — constant

`STAY` is prepended to the declaration. The value cannot be changed in any way.

```
STAY #pi = 3.14
STAY language = english
```

6.3 IF — reactive

`IF` is prepended. These variables can only contain booleans or operations that produce booleans. The prefix `#` is not necessary.

```
IF true_var = V          >> correct
IF error_var = 5         >> ERROR: 5 is not a boolean
```

The feature that makes them special: if an IF variable is set equal to another variable and that one changes, **the IF variable changes as well**.

```
#a = V
IF cond = @a
#a = F    >> now cond is also F
```

7 Console

The console is the space where the program interacts with the user.

7.1 TALK

`TALK` is the main keyword. Everything that follows is shown in the output.

```
hello = hello
TALK @hello world!
<* Output: hello world! *>
```

If `@` refers to a **non-existent** variable, the console automatically prompts for input and saves it in that variable:

```
TALK Hello, enter your details \nFirst name: @name \nLast name: @surname
TALK Hello @name @surname
```

7.2 OUT and INP

Variants of `TALK` considered syntactic sugar:

Keyword	Behaviour
`OUT`	Output only. With non-existent variables it raises an error instead of prompting for input.
`INP`	Requires at least one input.

Always use `TALK`. `OUT` and `INP` can be useful for clarity in certain contexts.

7.3 SCREAM

Prints a list showing the tags as well (unlike `TALK` which shows only the values):

```
list = [greeting | hello]
TALK @list    >> [hello]
SCREAM @list  >> [greeting | hello]
```

7.4 TALK:V and TALK:F — persistent input

`TALK:V` activates a persistent input that always appears at the end of the console output. The input entered is saved in the special variable `ASKED`.

```
TALK:V Enter a command:
TALK You typed: @ASKED

TALK:V New question:   >> overwrites the previous question

TALK:F                 >> deactivates the persistent input

TALK:V                 >> reactivates without a question
```

Warning: if an input is activated with `TALK` or `INP`, or no part of the code refers to `ASKED`, `TALK:V` will deactivate automatically.

7.5 Console text formatting

It is possible to colour and style text in the console using the escape sequence `\C{}`:

```
TALK \C{red+bold+_}Warning!\C{red}: \C{}greetings!
```

Syntax: `\C{colour+modifiers}`

Modifiers are combined with `+`. To restore the default colour use `\C{}` with empty braces. The style applies to all subsequent text up to the next `\C{}`.

Modifier	Effect
`bold`	Bold
`italic`	Italic
`-`	Strikethrough
`_`	Underline

Available named colours: `red`, `green`, `blue`, `yellow`, `orange`, `pink`, `cyan`, `purple`, `teal`, `white`, `gray`, `amber`, `lime`, `navy`, `gold`, `magenta`, `brown`, `sky`.

Hexadecimal colours are also supported: `\C{#ff9100}`.

```
TALK \C{red+bold+_}Warning!\C{red}: \C{}normal text
<* "Warning!" in bold red underlined,
   ":" in red, rest normal *>

TALK \C{#00e5ff}Hexadecimal cyan text\C{}
```

8 Mathematical operations

Mathematical operations take place only between numeric variables, inside a declaration or reassignment. Only the result is stored.

Operator	Operation
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`^`	Exponentiation
`/`	Division
`//`	Integer division (integer part of the result)
`%`	Remainder of division (modulo)

The `+` symbol also works between strings as concatenation.

9 Comparison operations

Always produce a boolean (`V` or `F`). They can be used inside any variable.

Operator	Meaning	Between
`==`	Equal	Any type
`!=`	Different	Any type
`>`	Greater than	Numbers only
`<`	Less than	Numbers only
`>=`	Greater than or equal to	Numbers only
`<=`	Less than or equal to	Numbers only

A numeric `5` is different from a string `5` — the type matters.

```
#var = 5
IF equal      = @var == 5          >> V
IF different  = @var != 3          >> V
IF greater    = @var > 3           >> V
IF type_check = \tvar == number    >> V
```

10 Logical operations

Operate only between booleans and produce a boolean. They can chain comparison operations.

Operator	Meaning
`ET`	AND — true if both are true
`VEL`	OR — true if at least one is true
`AUT`	XOR — true if exactly one is true
`!` (prefix)	NOT — inverts the boolean

```
#n = 7
IF between5and10 = @n > 5 ET @n < 10   >> V
IF outside       = @n < 5 VEL @n > 10  >> F
IF negated       = !@between5and10      >> F
```

11 Operator precedence

Precedence	Operators
1 (highest)	`^`
2	`*` `/`
3	`//`
4	`%`
5	`+` `-`
6	`==` `!=`
7	`<=` `>=`
8	`!`
9 (lowest)	`ET` `VEL` `AUT`

Parentheses `()` work as in mathematics to force a different order.

12 Incremental operators

Shortcuts to modify a numeric variable directly:

Operator	Equivalent to
var += n	var = @var + n
var -= n	var = @var - n
var *= n	var = @var * n
var /= n	var = @var / n
var //= n	var = @var // n
var %= n	var = @var % n
var ^= n	var = @var ^ n

```
#score = 10
#score += 5   >> now equals 15
#score *= 2   >> now equals 30
```

13 Functions

Functions are blocks of code that are defined without being executed. They are executed only when called.

13.1 FUN

```
FUN function_name(parameters)
  ...
  return value
  FEND
```

- `parameters` — external variables to pass to the function, separated by commas. There can be zero.
- `return` — optional. If included it must be at the end of the body. The return value can be saved or printed. Only one `return` statement per function.

```
FUN hello()
  TALK hello
  FEND
hello()
<* Output: hello *>

name = Mario
FUN greet(n)
  TALK hello @n!
  #num = 5
  return @num
  FEND
val = greet(name)
TALK @val
<* Output:
hello Mario!
5 *>
```

13.2 THEN

A simplified form of function: it takes no parameters and cannot have a `return`. It is closed with `THEND`.

```
THEN block_name
  ...
  THEND
```

Equivalent to `FUN name()` without `return`. THEN blocks can be called **only** inside a GO block.

14 GO Block

GO executes a function or a THEN block based on a boolean condition.

```
GO @cond_var @function_name #repetitions
```

- `@cond_var` — IF variable (or any boolean). The block is executed only if it is `V`.
- `#repetitions` — how many times to execute the block. If `1` it can be omitted. The special value `#c` repeats as long as the condition remains `V`.

```
IF cond = V
THEN block
  TALK Hooray!
  THEND
GO @cond @block       >> runs once

GO @cond @block #3    >> runs 3 times

#i = 0
IF loop = @i < 3
THEN count
  TALK @i
  #i += 1
  THEND
GO @loop @count #c   >> repeats while i < 3
```

14.1 Chains: `&` and `&&`

After a GO, alternative branches can be added:

- `&` — equivalent to `elif` in Python.
- `&&` — equivalent to `else` in Python, can only be placed at the end.

```
GO @cond1 @block1 & @cond2 @block2 && @block3
<* if cond1 → block1
   else if cond2 → block2
   else → block3 *>
```

14.2 Inline body with @{}

Instead of defining a separate THEN, the body can be written directly in the GO. The condition can also be inline:

```
GO @{@n > 0} @{
  TALK positive
} && @{
  TALK non positive
}
```

The special variable `THISGO` holds the current GO's condition and can be modified to interrupt the loop:

```
GO @{V} @{
  TALK @ASKED
  IF stop = @ASKED == done
  GO @stop @{ @THISGO = F }
}
```

15 Lists

A list stores multiple values in a single variable. Values are separated by `;` and enclosed in `[]`.

```
list = [hello ; hi]
```

Each element has an **index** starting from 0. Negative indices start from the right: `-1` is the last element.

Tags can be added to elements with `|`:

```
list = [greeting | hello ; hi]
list = [english | greeting | hello ; hi]  >> hello has two tags
```

To access an element:

```
list[#0]          >> by index
list[greeting]    >> by tag
list[tag1 ; tag2] >> by multiple tags
```

List values are always strings.

15.1 Adding elements

```
ADD list_name AT index = value   >> inserts, shifting the others forward
ADD list_name AT = value         >> appends at the end (index omitted)
ADD list_name BY index = value   >> replaces the element at that index
```

15.2 Removing elements

```
CANC list_name AT index             >> by index
CANC list_name BY tag               >> by tag
```

```
CANC list_name IS value             >> by value
CANC list_name IN [tag1 ; tag2]     >> by tag list
```

Warning: if two elements share the same tag or value, `CANC` removes both of them.

15.3 Printing a list

```
list = [greeting | hello]
TALK @list    >> [hello]
SCREAM @list  >> [greeting | hello]
```

Lists can contain other lists as elements:

```
data = [names | [Giulio ; Carlo] ; surnames | [Bianchi ; Ferrari]]
```

16 Variable scope

By default, variables declared inside a block are **local** — not accessible from outside.

Prefix	Effect
`RET`	Makes the variable global
`DEFRET`	Placed at the start of the block, makes **all** variables in the block global by default
`TEMP`	With `DEFRET` active, makes the single variable local

```
FUN compute()
  RET #result = 42   >> accessible everywhere
  local_v = hello    >> only inside this function
  FEND

FUN example()
  DEFRET
  #two = 2           >> global (DEFRET)
  TEMP #four = 4     >> local
  FEND
```

17 IN operator

Checks whether an element is present in a list. Produces a boolean.

```
list = [hello ; hi]
IF exists  = hello   IN list >> V
IF missing = goodbye IN list >> F
```

18 FOR loop

Used to iterate over the elements of a list.

```
FOR @list_name #[start ; end ; step] = @function()
```

- `start` — index to start from. Default: `0`.
- `end` — index to stop at. Default: `-1` (end of list).
- `step` — step between indices (positive or negative). Default: `1`.

All three values in `#[]` are optional; the `;` separator must still be included if only the middle one is omitted.

The following special variables are available inside the function:

Variable	Contents
`ONITEM`	The value of the current element
`ONINDEX`	The index of the current element
`ONTAG`	The tag(s) of the current element

```
nums = [10 ; 20 ; 30]
FOR @nums = @{TALK @ONINDEX: @ONITEM}
<* Output:
0: 10
1: 20
2: 30 *>

>> Reverse iteration
FOR @nums #[;;-1] = @{TALK @ONITEM}
<* Output: 30 / 20 / 10 *>
```

18.1 NUMBERS

The special list `NUMBERS` contains all real numbers, where each number has itself as its index. Useful for iterating numeric ranges:

```
FOR @NUMBERS #[1 ; 5] = @{TALK @ONITEM}
<* Output: 1 / 2 / 3 / 4 / 5 *>
```

19 Labels

Any statement can receive a label with the syntax `keyword:label`. To re-execute the statement associated with a label, use `*label`.

```
TALK:greeting Hello!
*greeting   >> re-runs the TALK:greeting
```

If more than one statement is assigned to the same label, `*label` executes the last one.

19.1 Special TALK labels

`TALK:V` — activates a persistent input at the end of the output. The input is saved in the special variable `ASKED` (see section 7).

`TALK:F` — deactivates the persistent input.

20 Error handling

20.1 Error codes

Code	Cause
VAR_NOT_FOUND	Non-existent variable
TYPE_ERROR	Invalid data type
INVALID_CONVERSION	Type conversion not possible
CONST_MODIFY	Attempt to modify a constant (STAY)
DIV_BY_ZERO	Division by zero
MATH_ERROR	Invalid mathematical operation
BOOL_EXPECTED	Boolean value expected
NUMBER_EXPECTED	Numeric value expected
STRING_EXPECTED	String expected
LIST_NOT_FOUND	Non-existent list
LIST_OUT_OF_RANGE	Index out of list bounds
LIST_EMPTY	Empty list
TAG_NOT_FOUND	Non-existent tag in the list
DUPLICATE_TAG	Duplicate tag not allowed
FUNC_NOT_FOUND	Non-existent function
INVALID_ARGUMENTS	Invalid arguments
RETURN_NOT_ALLOWED	Use of `return` not allowed
THEN_ARGUMENTS	Use of arguments in a THEN block
IF_NOT_FOUND	Non-existent IF block
GO_INVALID	Invalid GO block
GO_LOOP_ERROR	Infinite or invalid loop
INPUT_ERROR	Input error
OUTPUT_ERROR	Output error
SYNTAX_ERROR	Syntax error
ESCAPE_ERROR	Invalid escape
MOD_NOT_FOUND	Module not found
UNKNOWN_ERROR	Unknown error

20.2 TRY / SHOW / YET

Allow runtime errors to be handled without abruptly terminating the program.

```
TRY try_name
  ...code that might raise an error...
  TREND

SHOW show_name @try_name
  ...executed only if an error occurred...
  SEND

YET @try_name
  ...executed always (with or without an error)...
  YEND
```

`SHOW` and `YET` are optional.

Inside SHOW two special variables are available — they exist **only** there:

Variable	Type	Contents
ERRCODE	numeral	1 for all errors handled by LINE
ERRMSG	string	Descriptive error message

```
TRY trial
  OUT @nonexistent
  TREND

SHOW handle @trial
  TALK @ERRCODE
  TALK @ERRMSG
  SEND

YET @trial
  TALK Operation complete
  YEND
```

20.3 Rules

- TRY cannot be nested.
- Each TRY can have at most one SHOW and one YET.
- If an error occurs in a TRY without a SHOW, LINE prints ERRMSG and terminates execution.

21 Modules

Modules are collections of ready-to-use functions. They are imported with TAKE:

```
TAKE math
#r = math_sqrt(9)
TALK @r   >> 3
```

The functions of a module are called with the format modulename_functionname().

To see the list of all available modules, run .MODS in an empty file. To see the functions of a specific module, use .MATH, .STR, etc.

Module	Contents
`math`	Mathematical functions (sqrt, abs, floor, ceil, pow, sin, cos...)
`str`	String manipulation (len, sub, upper, lower, replace...)
`list`	List operations (len, has, sort, rev, shuffle...)
`rand`	Randomness (int, float, pick, shuffle, dice...)
`io`	Advanced input/output (time, date...)
`type`	Type checking and conversion
`conv`	Conversions (hex, bin, oct, tonum...)
`num`	Advanced numeric functions (fmt, fib, fact, gcd, prime...)

22 LINE files

A LINE program is saved with the `.line` extension and run by the interpreter.

23 LINE-DOM

LINE-DOM is a JavaScript layer that allows LINE to be used as a language for controlling the DOM of an HTML page, without writing JavaScript. It is included with the file `line-dom.js`.

```
<!-- LINE interpreter -->
<script src="interpreter.js"></script>
<!-- DOM bridge -->
<script src="line-dom.js"></script>
```

> **Fundamental rule:** inside `<script type="text/line">` tags goes *only* LINE code. Zero inline JavaScript.

23.1 Attributes of the `<script type="text/line">` tag

Attribute	Effect
`trigger="id"`	Runs the code on click on the element with that id
`autorun`	Runs the code when the page loads
`replace`	Overwrites the content of the target element instead of appending to it
`counter="id"`	Injects the special variable `__N__` (number of executions) and writes it to the indicated element
`body-class="id"`	Observes the indicated element and automatically applies its text as the `className` of `<body>` (useful for CSS themes)
`onclick="btnId:label,..."`	Binds the click of `#btnId` to the `label` block

```
<script type="text/line" trigger="btn-send" replace>
  reply = You clicked!
```

```
  <result>TALK @reply</result>
</script>

<button id="btn-send">Send</button>
<div id="result"></div>
```

23.2 Writing to HTML elements

LINE code is enclosed between tags bearing the id of the target element:

Syntax	Effect
`<id>TALK text</id>`	Replaces the element's content
`<id=after>TALK text</id>`	Appends to the end of the existing content
`<id=before>TALK text</id>`	Inserts at the beginning of the existing content
`<.class>TALK text</.class>`	Writes to all elements with that CSS class
`<.class=after>TALK text</.class>`	Appends to all elements with that class

The `\n` character inside DOM tags automatically becomes a `<br>` in the HTML (this does not apply to inputs).

```
<p id="greeting"></p>

<script type="text/line" trigger="btn" replace>
  <greeting>TALK Hello, world!</greeting>
</script>
```

23.3 Reading HTML input

LINE-DOM automatically injects the value of every `<input>`, `<textarea>` and `<select>` that has an `id` as a LINE variable before each execution. No declaration is needed.

```
<input id="name" type="text">
<button id="btn">Greet</button>
<div id="output"></div>

<script type="text/line" trigger="btn" replace>
  <output>TALK Hello, @name!</output>
</script>
```

Hidden inputs work as persistent memory between one execution and the next.

23.4 Handling clicks from LINE code

A click on an element can be registered directly in LINE syntax:

Syntax	Meaning
`THEN:btn-id blockname`	The THEN block is executed on click on that id
`GO:btn-id @{cond} @{...}`	Inline GO executed on click

```
<script type="text/line" autorun>
  THEN:btn-uppercase
```

```
    <output>TALK HELLO</output>
    THEND
</script>
```

> **Note:** `<id>` tags inside an inline `@{...}` body do not work. For blocks that write to elements, always use a named THEN.

23.5 LocalStorage with `<ls>`

LINE-DOM supports data persistence across page reloads via a special tag:

```
<ls>myvar = value</ls>
```

Saves `value` in localStorage with the key `line_myvar`. On each subsequent execution, the value is automatically injected as a LINE variable.

```
<input id="name">
<button id="btn-save">Save</button>
<div id="welcome"></div>

<script type="text/line" autorun>
  <welcome>TALK Hello, @name!</welcome>
</script>

<script type="text/line" trigger="btn-save">
  <ls>name = @name</ls>
</script>
```

23.6 Dynamic themes with `body-class`

The `body-class` attribute creates a `MutationObserver`: it observes the indicated element and automatically applies its text as a class of `<body>`. Combined with CSS variables, this allows themes to be changed without JavaScript.

```
<span id="theme" hidden></span>

<script type="text/line" trigger="btn" replace body-class="theme">
  <theme>TALK dark</theme>
</script>
```

```
body.dark  { --bg: #111; --fg: #eee; }
body.light { --bg: #fff; --fg: #111; }
```

23.7 Re-run behaviour

Every time the code is executed (for example on a trigger click), the entire script is re-executed from the beginning. This means that variables reset on each execution. For state persistence use `<ls>` (localStorage) or hidden inputs.

24 LINE-RENDER

LINE-RENDER is an IDE extension that allows mathematical formulas and chemical structures to be displayed in the console. It requires `line-render.js` and the KaTeX library.

The notation is an abbreviated form that is compiled into LaTeX before rendering.

24.1 RENDER blocks

Syntax for block rendering (on a separate centred line):

```
RENDER{
  formula
}
```

Everything between `{` and `}` is interpreted as a formula and rendered in the console.

```
RENDER{
  s(k=0,n)(k2) = n(n+1)/2
}

RENDER{
  de(f)/de(x) = lim(h -> 0)((f(x+h) - f(x))/h)
}
```

24.2 Inline render `\R...\R`

To include a formula directly within a line of text, use the escape pair `\R`:

```
TALK This is an integral \RS(0,1)(f(x))\R, and it is very interesting.
TALK Euler's formula is \Reu^(pi*i) + 1 = 0\R.
```

The formula between the two `\R` markers is rendered with KaTeX in inline mode, vertically aligned with the surrounding text.

24.3 CHEM mode — Chemical structures

To draw molecules the first line of the block must be `CHEM`. Each subsequent line describes a molecule.

```
RENDER{
  CHEM
  C(HHH=O)
}
```

Molecule syntax:

- Atoms: chemical symbols such as `C`, `H`, `O`, `N`, `Cl`, etc.
- Implicit bonds: `-` (single, default)
- Double bond: `=`
- Triple bond: `#`

- Branches: `(...)` for substituents

```
RENDER{
  CHEM
  C(HHH=O)          >> methanal (formaldehyde)
}

RENDER{
  CHEM
  C(H)(H)(=O)       >> carbonyl with two H
}
```

24.4 Mathematical notation

The notation is compact. Each expression is translated into LaTeX.

24.4.1 Basic operations

Notation	Rendering
`a + b`	$a + b$
`a - b`	$a - b$
`a * b`	$a \cdot b$
`a/b`	ab
`(a+b)/(c+d)`	$a + bc + d$
`n!`	$n!$

24.4.2 Exponent and subscript

Notation	Rendering
`x5`	x^5
`x^(n+1)`	x^{n+1}
`x#(i)`	x_i
`a#(i)^(2)`	a_i^2

24.4.3 Roots

Notation	Rendering
`R(x2+1)`	$\sqrt{x^2 + 1}$
`R3(a3)`	$\sqrt[3]{a^3}$

24.4.4 Derivatives

Notation	Type
`f'`, `y''`, `x'''`	Newton's notation
`de(f)/de(x)`	Ordinary derivative $df dx$
`dp(f)/dp(x)`	Partial derivative $\partial f \partial x$

24.4.5 Summations, products, integrals, limits

Notation	Rendering
`s(k=0,n)(k2)`	$\sum_{k=0}^{n} k^2$
`P(k=1,n)(k)`	$\prod_{k=1}^{n} k$
`S(0,1)(x2)`	$\int_{0}^{1} x^2\,dx$
`L(0)((sin(x))/x)`	$\lim_{x \to 0} \sin x x$

24.4.6 Standard functions

Notation	Function
`sin(x)`, `cos(x)`, `tan(x)`	Trigonometric
`arcsin(x)`, `arccos(x)`, `arctan(x)`	Inverse trigonometric
`sinh(x)`, `cosh(x)`, `tanh(x)`	Hyperbolic
`ln(x)`, `log(x)`, `exp(x)`	Logarithms and exponential

24.4.7 Vectors and matrices

Notation	Rendering
`v(x;y;z)`	Column vector
`M(2;2)(a;b;c;d)`	2×2 matrix

24.4.8 Combinatorics

Notation	Rendering
`C(n;k)`	nk
`Perm(n;k)`	A_n^k
`CombR(n;k)`	$n + k - 1k$

24.4.9 Absolute values and norms

Notation	Rendering
`|x - a|`	$\|x - a\|$
`||v||`	$\|\|v\|\|$

24.4.10 Decorators

Notation	Effect
`vc(v)`	Vector $\vec{v}$
`ha(x)`	Hat $\hat{x}$
`ba(x)`	Bar $\bar{x}$
`ti(x)`	Tilde $\tilde{x}$
`do(x)`	Dot $\dot{x}$
`do2(x)`	Double dot $\ddot{x}$
`fl(x)`	Floor $\lfloor x \rfloor$
`ce(x)`	Ceiling $\lceil x \rceil$

24.4.11 Constants and Greek letters

Notation	Symbol
`pi`	π
`eu`	e
`I`	∞
`na`	∇
`alp`, `bet`, `gam`, `del`, `eps`	$\alpha\ \beta\ \gamma\ \delta\ \varepsilon$
`the`, `lam`, `mu`, `sig`, `phi`, `ome`	$\theta\ \lambda\ \mu\ \sigma\ \varphi\ \omega$
`Gam`, `Sig`, `Phi`, `Psi`, `Om`	$\Gamma\ \Sigma\ \Phi\ \Psi\ \Omega$

24.4.12 Relations and logic

Notation	Symbol
`->`	$\rightarrow$
`=>`	$\Rightarrow$
`<=>`	$\Leftrightarrow$
`~`	$\approx$
`!=`	$\neq$
`<=`, `>=`	$\leq$, $\geq$

24.5 Console PDF export

The **PDF** button in the console toolbar generates a visual PDF of the console, preserving the rendering of formulas and chemical graphs. The PDF is resized to the actual content.

The **TXT** button instead saves the plain text of the console (formulas appear as source text).

24.6 HTML mode

The first line of the block can be `HTML`. All subsequent content is interpreted as raw HTML and inserted directly into the console via `innerHTML`.

```
RENDER{
  HTML
  ...HTML code...
}
```

This allows any HTML structure to be displayed in the console: tables, lists, images, styled text, arbitrary components.

```
RENDER{
  HTML
  <table border="1" style="border-collapse:collapse;color:#cdd9e5">
    <tr><th>Name</th><th>Value</th></tr>
    <tr><td>pi</td><td>3.14159</td></tr>
    <tr><td>e</td><td>2.71828</td></tr>
  </table>
}

RENDER{
```

```
  HTML
  <ul style="color:#80cbc4">
    <li>First element</li>
    <li>Second element</li>
  </ul>
}
```

> **Note:** the HTML content is inserted without sanitisation. Using script tags inside is not recommended — the HTML is rendered in the context of the IDE console.