# LogoMor Documentation

## What is Logo

Logo is a programming language that dictates the drawing of a sketch.

The drawing vehicle is an avatar that can be moved inside the 3-dimentional space, leaving a trail behind it. The avatar is initiated at the origin with orientation towards the y axis.

Movement commands indicate the movement of the avatar with reference to its current position and orientation. Exceptionally, the set commands can place the avatar to a specific position with reference to the absolute coordinate system of the space.

## Movement Commands

These are the basic commands that control the movement of the avatar.

All following commands do not return any value

|  |  |  |  |
| --- | --- | --- | --- |
| Command Name | Arguments | Description | Example |
| fd or forward | 1 (steps distance) | Move forwards | fd 50 |
| bk or back | 1 (steps distance) | Move backwards | bk 50 |
| rt or right | 1 (degrees) | Turn right | rt 90 |
| lt or left | 1 (degrees) | Turn left | lt 90 |
| up | 1 (degrees) | Turn up | up 90 |
| dn or down | 1 (degrees) | Turn down | dn 90 |
| rr or roll\_right | 1 (degrees) | Roll right | rr 90 |
| rl or roll\_left | 1 (degrees) | Roll left | rl 90 |
| home | - | Move to home position | home |
| setx | 1 (position) | Move to specific x coordinate | setx 50 |
| sety | 1 (position) | Move to specific y coordinate | sety 50 |
| setz | 1 (position) | Move to specific z coordinate | setz 50 |
| setxyz | 3 (position) | Move to specific point (x,y,z) | setxyz 50 50 50 |
| arc | 2 (angle degrees,  radius) | Draw a clockwise arc with center the avatar position. Avatar does not move | arc 90 50 |

## Position Commands

These are the commands that provide information about the current position of the avatar in the 3 dimensional space.  
Their return value can be used by any Logo command accepting arguments

|  |  |  |  |
| --- | --- | --- | --- |
| Command Name | Arguments | Description | Example |
| getx | - | returns the current position on x axis | getx |
| gety | - | returns the current position on y axis | gety |
| getz | - | returns the current position on z axis | getz |
| dist | 3 (x, y, z) | returns the distance of the current position from a specified point in space | dist 20 35 40 |

## Trail appearance Commands

These are the commands that configure whether the avatar movement leaves a trail and how it should appear. Once called, they affect all consecutive movement commands until another change is made.

All following commands do not return any value

|  |  |  |  |
| --- | --- | --- | --- |
| Command Name | Arguments (unit) | Description | Example |
| penup or pu | - | Avatar stops leaving trail | penup |
| pendown or pd | - | Avatar starts leaving tail | pendown |
| setpensize or sps | 1 (pixels) | Sets the trail width | setpensize 4 |
| color | 3 (0-255 r g b) | Sets the trail color in RGB | color 255 20 40 |
| colorhsb | 3 (0-360 h, 0-100 s, 0-100 b) | Sets the trail color in HSB | color 300 90 90 |
| coloralpha | 1 (0-255) | Sets the color opacity | coloralpha 128 |
| showturtle or st | - | Shows the avatar (default) | showturtle |
| hideturtle or ht | - | Hides the avatar | hideturtle |

## Output Commands

These are the commands that enable output to the user

All following commands do not return any value

|  |  |  |  |
| --- | --- | --- | --- |
| Command Name | Arguments (unit) | Description | Example |
| print | 1 | Prints a value to the terminal | print “starting |
| label | 1 | Displays a value on the sketch | label “corner |
| settextsize or sts | 1 (pixels) | Sets the font size used by label | settextsize 15 |
| point | - | Draws a point at current position. | point |

## Arguments

An argument in Logo can be:

* Any number. E.g. 2, 3.14, -75.1
* The value of a defined variable name, using the prefix ‘:’ E.g. fd :var
* A word literal, using the prefix ‘”’. Only a few commands can accept this kind of argument

E.g. print “helloWorld label “corner

* Any function or command that returns a value e.g. print getx
* Any expression with a combination of arguments and arithmetic operators (+,-,\*,/) or comparison operators (<,>,<=,>=,=) .

E.g. fd :n + 10 rt 360 / 5 print :k <= 5

The result of a comparison is 1 if the comparison is true and 0 if it is false

##### Note 1:

A literal can also be constructed by string concatenation of 2 arguments, using the command word

E.g. print word “hello “world

print word “item 1

##### Note 2:

A space character can be encoded into a literal for using with output commands, with the sequence ‘\s’

E.g. print “hello\sworld This will print ‘hello world’

##### Note 3:

Any expression can be sub-grouped using parentheses ‘()’\*

**\***Parentheses are useful for determining the precedence of operations. Moreover, they are critical for separating arguments in the case of negative numbers.

E.g.: To set the position of the avatar to x:6, y:-7, z:8, Writing the below:

Setxyz 6 -7 8

Will yield an error, as it will consider that the first argument is 6-7 and the second argument is 8. The error will indicate that the command is missing a third argument. The correct way is to write:

setxyz 6 (-7) 8.

Grouping like that leaves no ambiguity for the arguments separation. Of course parentheses may be optionally used for the other arguments as well.

##### Note 4:

Operators precedence: multiplication/division > addition/subtraction > comparisons > equality

Associativity among operations of equal precedence: Left-to-Right

## Variables

Variables are places in memory that we can store any numeric value or literal.

The variables can be assigned names and values by the user, using the make command

After setting it, the value of that variable can be accessed by using the prefix ‘:’ followed by its name.

Alternatively, we can get the value of a variable by using command thing, followed by the variable’s name literal

Any variables made outside of a function are considered ‘global’ variables and can be accessed by any part of the code, as long as they have already been made when the reference occurs.

Variables made in a function are considered local and can only be accessed within that function.

There is no difference when defining a variable or assigning a value to it. Using make, if the variable already exists at the current scope or in global or static scope, it is assigned a new value. Otherwise, it is created.

##### Syntax:

make <variable name literal> <variableValue>

:< variable name >

##### Example :

Make “var 50

fd :var (now equivalent to fd 50)

fd thing “var (same as above)

\*Note1: we can increment/decrement variables by using commands increment, decrement followed by the variable name literal:

increment “var (equivalent to make “var :var + 1)

decrement “var (equivalent to make “var :var - 1)

\*Note2: Storing variable names literals in variables provide a means of referencing variables and passing variables by reference to functions

##### Example:

make “a 5

make “b “a

print “b ;will print ‘b’

print thing “b ;will print ‘a’

print thing thing “b ;will print ‘5’

## Static Variables

Regular variables may have global or local scope, but their lifetime is limited to a single execution of the program. Using the keyword static, the user can create a variable that is initiated only the first time that the interpreter will come across it in the source code, and the variable remains accessible for the whole lifetime of the program. After initiating the static variables, the user can access them regularly with the ‘:’ prefix and assign new values to it with the make keyword

##### Syntax:

static <variable name literal><initializationValue>

##### Example:

static “counter 0

make “counter :counter+1

print :counter

The above example will print an incrementing number on each frame/execution, starting from 1

## Conditionals

### If, Ifelse

A block of code can be executed conditionally, by using the command if

##### Syntax:

If <condition> [ <commands to execute if condition is true> ]

<condition>: An argument expression that can be evaluated as true or false

< commands to execute if condition is true > : any program code

##### Example:

If :n < 5 [ print “lessThanFive ]

Similarly, two different blocks of code can be executed, depending on the truth value of a condition, using ifelse

##### Syntax

Ifelse <condition > [ <commands to execute if condition is true> ] [ <commands to execute if condition is false> ]

##### Example

Ifelse :n < 5 [ print “lessThanFive ] [ print “higherOrEqualToFive ]

Note:

Any plain number can also be evaluated for its truth value. Any value other that 0 evaluates as true and the value of 0 evaluates as false.

E.g if 5 [ fd 10 ] - The fd command will be executed

If 0 [ fd 10 ] - The fd command will not be executed

## Loops

### Repeat

A block of code can be executed repeatedly n times, with the command repeat:

##### Syntax:

Repeat <number of executions> [ <commands> ]

##### Example:

repeat 4 [ fd 10 rt 90 ]

##### Note:

Using the command repcountwe can get the number of the current execution of the innermost repeat block being executed, starting from number 1. If no repeat block is executed, 0 is returned

### While

A block of code can be executed repeatedly, as long as a condition is true, with the command while

##### Syntax:

While <condition> [ <commands>]

##### Example:

make “n 4

while :n > 0 [ fd 10 rt 90 make “n :n – 1 ]

### Until

Similarly, a block of code can be executed repeatedly, as long as a condition is false, with the command until

##### Syntax:

until <condition> [ <commands>]

##### Example:

make “n 4

until :n = 0 [ fd 10 rt 90 make “n :n – 1 ]

## Functions

* A function is a part of code (called function’s body) that can be given a name and can be executed whenever this name is called inside the program.
* A function can accept any predefined number of parameters and use them inside its body as variables. The declaration of the parameters names is done by using the prefix ‘:’
* A function can optionally return a value to the command that called it. The return statement can be at any position in the body and the execution will stop once it reaches it
* Functions are called by using their names, followed by their parameters arguments
* The function parameters and variables declared inside the function body define a ‘local scope’ of variables, visible only within the function. If there is also a global variable with the same name, the local variable takes priority.
* ***In fact, all Logo commands can be considered as functions***

##### Syntax:

to <functionName> <list of parameter names> <body> <return statement> end

##### Example:

to square :side repeat 4 [ fd :side rt 90 ] end

to add :a :b return :a + :b end

square 50 – will draw a square of side length 50

print add 2 3 – will print the number ‘5’ on the terminal

square add 10 40 – will first call function add with parameters 10, 40 and then call square with parameter the output of function add, which will be 50

## Mathematical commands

These are commands that are useful for performing mathematical calculations

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Arguments | Return value units | Description | Example |
| sqrt | 1 |  | compute square root | sqrt 4 |
| pow | 2 (base, exponent) |  | raises the base to the exponent | pow 2 3 |
| mod | 2 (Divisor, divider) |  | remainder of integer division | mod 4 3 |
| cos | 1 (degrees) |  | cosine of angle | cos 60 |
| sin | 1 (degrees) |  | sine of angle | sin 30 |
| tan | 1 (degrees) |  | tangent of angle | tan 30 |
| arccos | 1 | degrees | inverse cosine | arccos 0.5 |
| arcsin | 1 | degrees | inverse sine | arcsin 0.5 |
| arctan | 1 | degrees | inverse tangent | arctan 4 |
| degtorad | 1 | radians | convert degrees to radians | degtorad 180 |
| radtodeg | 1 | degrees | convert radians to degreed | radtodeg pi |
| ln | 1 |  | natural logarithm | ln 7 |
| log | 1 |  | logarithm with base 10 | log 150 |
| exp | 1 |  | e raised to value | exp 2 |
| pi | 0 |  | returns the number π | pi |
| int | 1 |  | rounds to closest integer | int 7.8 |
| trunc | 1 |  | truncates integer part | trunc 7.8 |
| abs | 1 |  | returns absolute value | abs -3.7 |
| min | 2 |  | returns smallest of two values | min 3 5 |
| max | 2 |  | returns biggest of two values | max 3 5 |

## Logical commands/functions

These are commands that are useful for performing logical operations between arguments that can be evaluated for their truth value

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Arguments | Description | Example |
| or | 2 | Returns true if any of the arguments is true | or :n < 5 :n > 10 |
| and | 2 | Returns true if both of the arguments are true– | and :n < 5 :k < 5 |
| not | 1 | Returns true if the argument is false | not :n = 5 |

## Random number generation

The following command is useful to produce random numbers

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Arguments | Description | Example |
| rand | 1 | Returns an integer random number in the range of [0, n) where n is the argument\* | rand 100 |

\*Since the drawing is repeated in every frame of the display, the same random number will be returned at every execution, but a different one for each new run of the program. Alternatively, use **randcrazy** for a different output at each frame

## Timing commands

These are commands that provide timing information and enable the user to create animated sketches

Explanation: The sketch code is run over and over again and displayed multiple times per second. Each execution displays a ‘frame’ to the view panel. All code and variables remain the same for every execution, except these timing functions, that provide timing info starting from the first run of the program. This means that they give a different result at every frame. And this ability can be exploited to produce animated sketches. None of the below commands accept any arguments

|  |  |  |
| --- | --- | --- |
| Name | Description | Example |
| time | Returns the current time in seconds since the first run of the program | time |
| frame | Returns the current frame number since the first run of the program | frame |

## Faces

These are commands that enable the user to draw surfaces as solid objects faces. To draw a face the user should begin it on its first vertex, draw the edges of the face by using the regular move commands and then end it on its final vertex. When ending a face the avatar should have arrived back to the initial vertex, but if not, the face will be closed with an extra edge to the initial vertex. This extra edge will not not create a line, to indicate the opening to the user.

|  |  |  |
| --- | --- | --- |
| Name | Description | Example |
| beginface | Starting a new surface on its initial vertex | beginface |
| endface | Ending a surface on its final vertex | endface |

## Mouse Commands

The following commands provide info about the user’s mouse position and button.

The position of the mouse is 2D and it is calculated with reference the origin of the drawing.

The position calculation takes into account the pan and zoom of the drawing, but not the rotation.

The viewing perspective also affects the position calculation.

All of the below commands take no arguments

|  |  |  |
| --- | --- | --- |
| Name | Description | Example |
| mousex | Returns the X position of the mouse | mousex |
| mousey | Returns the Y position of the mouse | mousey |
| mousepressed | Returns true if the mouse button is pressed, false otherwise | mousepressed |

## 3D Primitives

These are the commands that draw some basic 3D shapes.

The shapes are solid objects with faces, centered at the avatar’s position.

If the avatar’s tail is visible (pd command), the strokes that produce the shapes are also visible.

All of the following commands do not return any value

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Arguments | Description | Example |
| box | 1 (side length) | Draws a square box | box 50 |
| sphere | 1 (radius) | Draws a shere | sphere 50 |
| cylinder | 2 (radius, height) | Draws a cylinder | cylinder 30 50 |
| cone | 2 (radius, height) | Draws a cone | cone 30 50 |
| torus | 2 (radius, tube radius) | Draws a torus | torus 30 50 |
| ellipsoid | 3 (radiusX, radiusY, radiusZ) | Draws an ellipsoid | ellipsoid 30 50 20 |

## Models

By pressing the ‘Add’ button of the MODELS area, we can upload valid .STL or .OBJ 3D object files

The objects described in the files are loaded as models and they are given initial default names (model1, model2 etc.). These names can be changed by the user to any valid literal.

The models can be drawn on the sketch using the model command, followed by the model name we want to draw and the size of it. The size is defined as the side length of a cube that would exactly fit the object. When the interpreter encounters a model command, it renders the corresponding object centered at the avatar’s current position. If the avatar’s tail is visible (pd command), the strokes that produce the object are also visible.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Arguments | Description | Example |
| model | 2 (model name literal, size) | Draws the object on the sketch | model “model1 50 |

## Images

By pressing the Add button of the IMAGES area, we can upload image files (jpg, png, gif). The loaded images are given initial names (image1, image2, etc.) These names can be changed by the user to any valid literal. The images can be drawn on the sketch by using the image command, followed by the image name and the desired image height. The width of the image is automatically calculated so that the aspect ratio of the original image is not altered. The top-left corner of the image will appear on the current avatar position and the plane of the image will be the avatar’s X-Y plane.

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Arguments | Description | Example |
| image | 2 (image name literal, height) | Draws the image on the sketch | image “image1 50 |

## Sounds

By pressing the Add button of the SOUNDS area, we can upload audio files (mp3. wav, etc).

The loaded sounds are given initial names (sound1, sound2, etc.) These names can be changed by the user to any valid literal. Next to the uploaded sounds the user can find the controls for each of them and manipulate them (play/pause, adjust volume etc). The user can also control the flow of these audio files programmatically, with the sound\_play, sound\_stop and sound\_pause commands, followed by the sound name literal. Note that these commands override the corresponding screen controls. Note that the play command will start playing the audio file if it was previously paused, stopped or ended. So if a play command is executed repeatedly, the corresponding sound will be played in a loop. With the rest of the sound commands the user can control/check the playing status, current playing second and volume of a sound.

#### Note:

The Logomor programs are executed repeatedly in a loop. But the sound commands imply their own timing. The user must take this into account when coding. E.g. the command sound\_settime in a loop would keep a sound steady on the same timestamp and the sound would not be heard.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Arguments | Returns | Description | Example |
| sound\_play | 1 (sound name) | - | Play/Resume | sound\_play “sound1 |
| sound\_stop | 1 (sound name) | - | Stop | sound\_stop “sound1 |
| sound\_pause | 1 (sound name) | - | Pause | sound\_pause “sound1 |
| sound\_isplaying | 1 (sound name) | 1 or 0 | 1 if the sound is currently playing | sound\_isplaying “song |
| sound\_gettime | 1 (sound name) | sec | Returns current sec of sound | sound\_gettime “song |
| sound\_settime | 2(name, sec) | - | Sets the current sec of sound | sound\_settime “song 2.5 |
| sound\_getvolume | 1 (sound name) | 0-100 | Returns the volume setting | sound\_getvolume “song |
| sound\_setvolume | 2(name, 0-100) | - | Sets the volume | sound\_setvolume “song 60 |

## Comments

Comments are notes on the source code that are used for human readability and are not part of the actual program.

Comments in Logo start with the semicolon ‘;’. Any appearance of the semicolon will make the interpreter ignore the rest of the specific line where it appeared

##### Example

fd 20 ;move a bit forward

## Ignored Characters

Space characters are vital to separate keywords, function names, variable names and arguments.  
The user can add as many of these characters in-between the source code tokens as desired, without any effect on the actual code.

These characters are: space( ), new line(\n), carriage return(\r), tab(\t), vertical tab(\v), form feed(\f)

## Naming

* The interpreter treats the source code in a case-insensitive manner. This means that capitalization does not alter the way a token is conceived. E.g., EnD is the same as end
* Variable names and function names may contain any sequence of alphanumeric characters. Symbols as any of the operators (+,-,\*,/,<,>,=), parentheses or brackets or ‘;’ are not allowed.
* The symbols mentioned above also cannot be part of literals used with the prefix “