KTurtle's Logo Programming Reference Commands

Using commands you tell the turtle or KTurtle to do something. Some commands need input, some give output. In this section we explain all the commands that can be used in KTurtle. Please note that all build in commands we discuss here are highlighted with dark green in the code editor, this can help you to distinguish them.

1. Moving the turtle There are several commands to move the turtle over the screen	
forward (fw) forward X	forward moves the turtle forward by the amount of X pixels. When the pen is down the turtle will leave a trail. forward can be abbreviated to fw
backward (bw) backward X	backward moves the turtle backward by the amount of X pixels. When the pen is down the turtle will leave a trail.
turnleft (tl) turnleft X	turnleft commands the turtle to turn an amount of X degrees to the left
turnright (tr) turnright X	turnrightthe turtle to turn an amount of X degrees to the right.
direction (dir) direction X	direction set the turtle's direction to an amount of X degrees counting from zero, and thus is not relative to the turtle's previous direction.
center	center moves the turtle to the center on the canvas
go go X,Y	go commands the turtle to go to a certain place on the canvas. This place is X pixels from the left of the canvas, and Y pixels form the top of the canvas. Note that using the go command the turtle will not draw a line
gox gox X	gox using this command the turtle will move to X pixels from the left of the canvas whilst staying at the same height.
goy goy Y	goy using this command the turtle will move to Y pixels from the top of the canvas whilst staying at the same distance from the left border of the canvas

2. The turtle has a pen The turtle has a pen that draws a line when the turtle moves. There are a few commands to control the pen. In this section we explain these commands	
penup (pu)	penup lifts the pen from the canvas. When the pen is “up” no line will be drawn when the turtle moves. See also pendown
pendown (pd)	pendown presses the pen down on the canvas. When the pen is press “down” on the canvas a line will be drawn when the turtle moves.
penwidth (pw) penwidth X	penwidth sets the width of the pen (the line width) to an amount of X pixels
pencolor (pc) pencolor R,G,B	pencolor sets the color of the pen. pencolor takes an RGB combination as input

3. Commands to control the canvas	
canvassize (cs) canvassize X,Y	With the canvassize command you can set the size of the canvas. It takes X and Y as input, where X is the new canvas width in pixels, and Y is the new height of the canvas in pixels
canvascolor (cc) canvascolor R,G,B	canvascolor set the color of the canvas. canvascolor takes an RGB combination as input
wrapon	With the wrapon command you can set wrapping “on” for the canvas. Please see the glossary if you want to know what wrapping is
wrapoff	With the wrapoff command you can set wrapping “off” for the canvas: this means the turtle can move off the canvas and can get “lost”.

4. Commands to clean up There are two commands to clean up the canvas after you have made a mess	
clear (cr)	With clear you can clean all drawings from the canvas. All other things remain: the position and angle of the turtle, the canvascolor, the visibility of the turtle, and the canvas size
reset	reset cleans much more thoroughly than the clear command. After a reset command everything is like is was when you had just started KTurtle. The turtle is positioned at the middle of the screen, the canvas color is white, and the turtle draws a black line on the canvas

5. The turtle is a sprite First a brief explanation of what sprites are: sprites are small pictures that can be moved around the screen, like we often see in computer games. Our turtle is also a sprite. For more info see the glossary on sprites. Next you will find a full overview on all commands to work with sprites. 6. [The current version of KTurtle does not yet support the use of sprites other than the turtle]	
show (ss)	show makes the turtle visible again after it has been hidden
hide (sh)	hide hides the turtle. This can be used if the turtle does not fit in your drawing

7. Can the turtles write? The answer is: “yes”. The turtle can write: it writes just about everything you command it to	
print print X	The print command is used to command the turtle to write something on the canvas. print takes numbers and strings as input. You can print various numbers and strings using the “+” symbol. See here a small example: año = 2010 autor = "María" print autor + " en el " + año + " cumplirá 18 años "
fontsize fontsize X	fontsize sets the size of the font that is used by print. fontsize takes one input which should be a number. The size is set in pixels

8. A command that rolls dice for you

There is one command that rolls dice for you, it is called random, and it is very useful for some unexpected results.

random random X,Y	random is a command that takes input and gives output. As input are required two numbers, the first (X) sets the minimum output, the second (Y) sets the maximum. The output is a randomly chosen number that is equal or greater then the minimum and equal or smaller than the maximum. Here a small example: repeat 500 [x = random 1,20 forward x turnleft 10 - x]

Using the random command you can add a bit of chaos to your program.Input and feedback though dialogs. A dialog is a small pop-up window that provides some feedback or asks for some input.

9. KTurtle has two commands for dialogs, namely: message and ask

message message X	The message command takes a string as input. It shows a pop-up dialog containing the text from the string. year = 2010 author = "María" print author + " en el " + year + " cumplirá 18 años "

ask
 	
ask X
	ask takes a string as input. It shows this string in a pop-up dialog (similar to message), along with an input field. After the user has entered a number or a string into this, the result can be stored in a variable or passed as an argument to a command. For example:
	`$in = ask "What is your year of birth?"`
	`$out = 2003 - $in`
	`print "In 2003 you were " + $out + " years old at some point."`

The user cancels the input dialog, or does not enter anything at all, the variable is empty

10. Assignment of variables

First we have a look at variables, then we look at assigning values to those variables.

Variables are words that start with a "$", in the editor they are *highlighted* with purple.

Variables can contain any number, string or boolean (true/false) value. Using the assignment, **=**, a variable is given its content. It will keep that content until the program finishes executing or until the variable is reassigned to something else.

You can use variables, once assigned, just as if they are their content. For instance in the following piece of TurtleScript:

```
$x = 10
$x = $x / 3
print $x
```

First the variable **$x** is assigned to **10**. Then **$x** is reassigned to itself divided by **3** — this effectively means **$x** is reassigned to product of **10 / 3**. Finally **$x** is printed. In line two and three you see that **$x** is used as if it is its contents.

Variables have to be assigned in order to be used. For example:

```
print $n
```

Will result in an error message.

Please consider the following piece of TurtleScript:

```
$a = 2004
$b = 25
# the next command prints "2029"
```

10. Assignment of variables

```
print $a + $b
backward 30
# the next command prints "2004 plus 25 equals 2029"
print $a + " plus " + $b + " equals " + ($a + $b)
```

In the first two lines the variables **$a** and **$b** are set to 2004 and 25. Then in two **print** commands with a **backward 30** in between are executed. The comments before the **print** commands explain what they are doing. The command **backward 30** is there to make sure every output is on a new line. As you see variables can be used just as if their where what they contain, you can use them with any kind of operators or give them as input when invoking commands.

One more example:

```
$name = ask "What is your name?"
print "Hi " + $name + "! Good luck while learning the art of programming..."
```

Pretty straight forward. Again you can see that the variable **$name**, treated just like a string.

When using variables the inspector is very helpful. It shows you the contents of all variables that are currently in use.

11. Controlling execution

The execution controllers enable you — as their name implies — to control execution. Execution controlling commands are *highlighted* with dark green in a bold font type. The brackets are mostly used together with execution controllers and they are *highlighted* with black.

wait

If you have done some programming in KTurtle you have might noticed that the turtle can be very quick at drawing. This command makes the turtle wait for a given amount of time.

wait makes the turtle wait for X seconds.

```
repeat 36 {
  forward 5
  turnright 10
  wait 0.5
}
```

`wait X`

This code draws a circle, but the turtle will wait half a second after each step. This gives the impression of a slow-moving turtle.

if

The code that is placed between the brackets will only be executed `if` the boolean value evaluates “true”.

```
$x = 6
if $x > 5 {
  print "$x is greater than five!"
}
```

`if boolean { ... }`

On the first line $x is set to 6. On the second line a comparing operator is used to evaluate $x > 5. Since this evaluates “true”, 6 is larger than 5, the execution controller `if` will allow the code between the brackets to be executed.

11. Controlling execution

The execution controllers enable you — as their name implies — to control execution. Execution controlling commands are *highlighted* with dark green in a bold font type. The brackets are mostly used together with execution controllers and they are *highlighted* with black.

Else

`if boolean { ... } else { ... }`

`else` can be used in addition to the execution controller `if`. The code between the brackets after `else` is only executed if the boolean evaluates “false”.

```
reset
$x = 4
if $x > 5 {
  print "$x is greater than five!"
} else {
  print "$x is smaller than six!"
}
```

The comparing operator evaluates the expression `$x > 5`. Since 4 is not greater than 5 the expression evaluates “false”. This means the code between the brackets after `else` gets executed.

while

`while boolean { ... }`

The execution controller **`while`** is a lot like **`if`**. The difference is that **`while`** keeps repeating (looping) the code between the brackets until the boolean evaluates “false”.

```
$x = 1
while $x < 5 {
  forward 10
  wait 1
  $x = $x + 1
}
```

On the first line **`$x`** is set to 1. On the second line **`$x < 5`** is evaluated. Since the answer to this question is “true” the execution controller **`while`** starts executing the code between the brackets until the **`$x < 5`** evaluates “false”. In this case the code between the brackets will be executed 4 times, because every time the fifth line is executed **`$x`** increases by 1.

repeat

`repeat number { ... }`

The execution controller `repeat` is a lot like `while`. The difference is that `repeat` keeps repeating (looping) the code between the brackets for as many times as the given number.

for

`for variable = number to number { ... }`

The `for` loop is a “counting loop”, it keeps count for you. The first number sets the variable to the value in the first loop. Every loop the number is increased until the second number is reached.

```
for $x = 1 to 10 {
  print $x * 7
  forward 15
}
Every time the c
```

Every time the code between the brackets is executed the `$x` is increased by 1, until `$x` reaches the value of 10. The code between the brackets prints the `$x` multiplied by 7. After this program s execution you will see the times table of 7 on thef inishes it canvas.

break

Terminates the current loop immediately and transfers control to the statement immediately following that loop.

exit

Finishes the execution of your program.