Processing

- Logic
- Loops
- Mathematics
- Text
- Data structures
- Util
- Variables
- Functions
- Machine Learning
- Imports

Logic

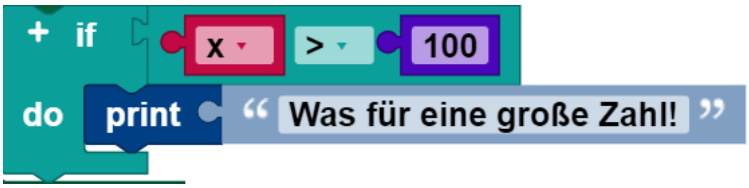
Conditional statements

Conditional instructions are essential for programming. They make it possible to formulate case differentiations, such as:

- If there is a path to the left, turn left.
- If the number of points = 100, press “Good job!”.

if blocks

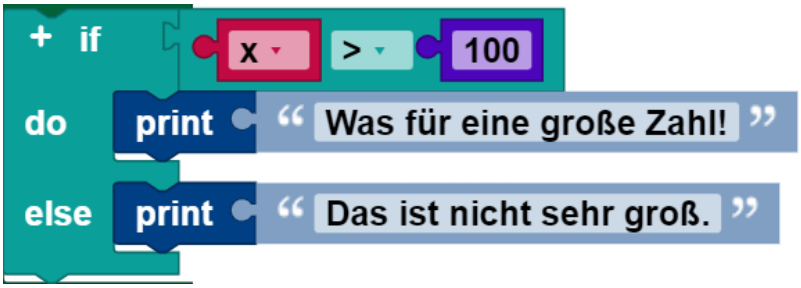
The simplest condition is an **if** block:



When it is executed, the value of the variable **x** is compared to 100. If it is larger, then “What a large number!” is output. Otherwise, nothing happens.

if else blocks

It is also possible to indicate that something should happen when the condition is false, as in this example:

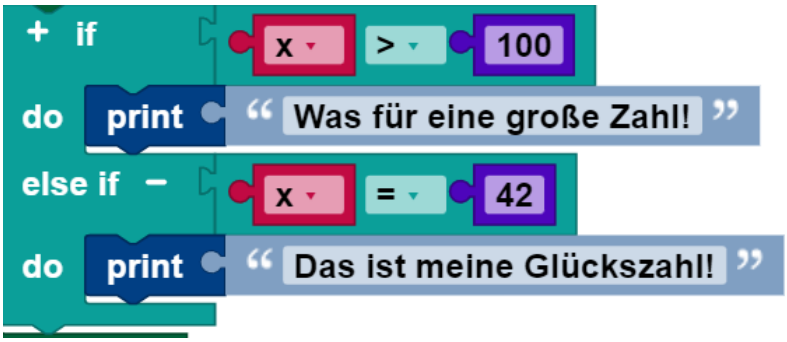


As in the previous block, “What a large number!” is output when **x** > 100. Otherwise “It's not very big’ is output.

An **if** block may have a **do** section, but not more than one.

if do else if blocks

It is also possible to test multiple conditions with a single **if** block, by adding **do else** clauses:

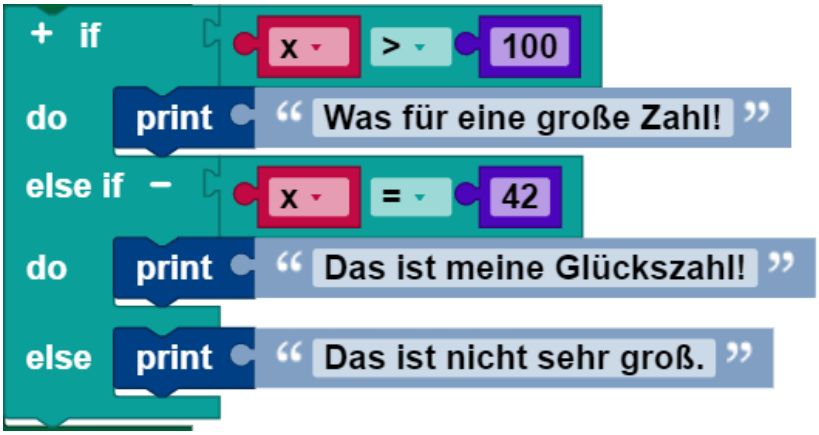


The block checks first whether **x** > 100, and outputs “What a large number!” if this is the case. If this is not the case, it then checks whether **x** = 42. If so, then it outputs “That is my lucky number!”. Otherwise, nothing happens.

An **if** block can have any number of **if do** sections. The conditions are evaluated from top to bottom, until one of them is fulfilled, or until there are no more conditions left.

if do else if do else blocks

if blocks can have both **if do** and **else if** sections:

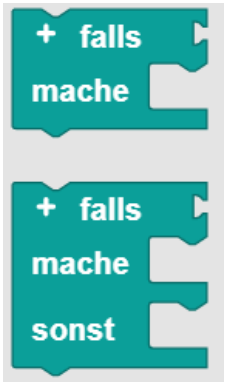


The **else if** section guarantees that an action is executed, even if none of the previous conditions is true.

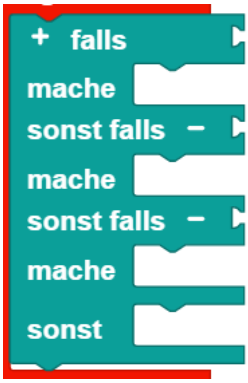
An **else if** section can also occur after any number of **if do** sections, including zero, which would then be a completely normal **if do** block.

Block modification

Only the simple **if** block and the **if do** block appear in the tool list:



To add **if do** and **else** clauses, click the (+) symbol. The (-) symbol can be used to remove **else if** clauses:



Note that the shapes of the blocks permit any number of **else if** sub-blocks to be added, but only up to one **if** block.

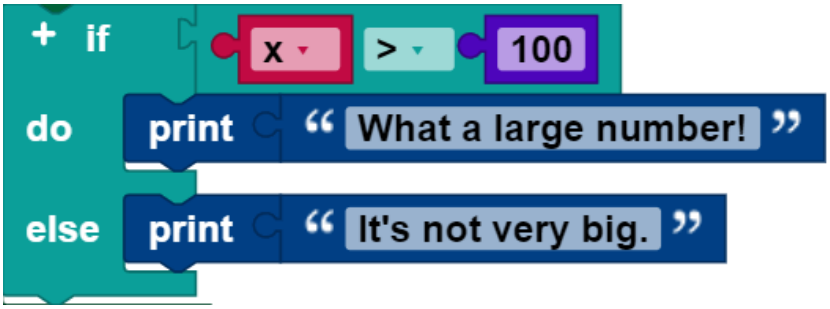
Boolean logic

Boolean logic is a simple mathematical system with two values:

- **true**
- **incorrect**

Logic blocks in ROBO Pro Coding are generally there to control conditions and loops.

Here is an example:



If the value of x is not greater than 100, then the condition is false, and “It’s not very big.” is output. If the value of x is not greater than 100, then the condition is false and “It’s not very big.” is output. Boolean values can also be saved in variables and transmitted to functions, just like numbers, texts, and list values.

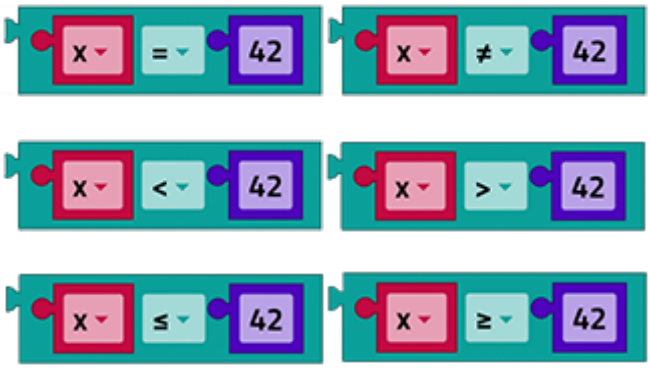
If a block expects a Boolean value as an input, then no input will be interpreted as **false**. Non-Boolean values cannot be inserted directly where Boolean values are expected, although it is possible (but not advisable) to save a non-Boolean value in a variable and then insert this into the condition input. This method is not recommended, and its behavior can change in future versions of ROBO Pro Coding.

Values

An individual block with a drop down list that either indicates **true** or **false** can be used to access a Boolean value:

Comparative operators

There are six comparative operators. Two inputs are entered into each (normally two numbers), and the comparative operator returns **true** or **false**, depending on how the inputs are compared to one another.



The six operators are: equal, not equal, less than, greater than, less than or equal, greater than or equal.

Logical operators

The **and** block returns **true** if and only if its two input values are true.

The **or** block returns **true** if at least one of its two input values is true.

do

The **not** block converts a Boolean input into its opposite. For example, the result of:

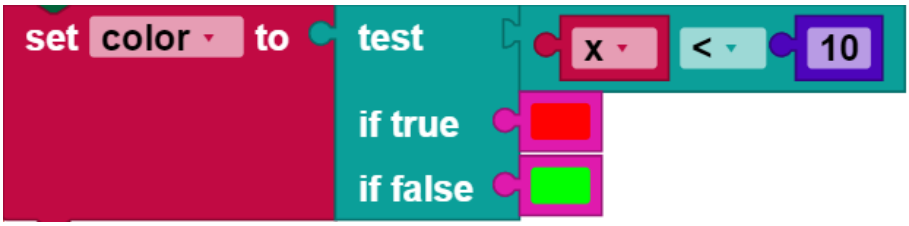
is **false**.

If there is no input, then the value **true** is assumed, so that the following block will generate the value **false**:

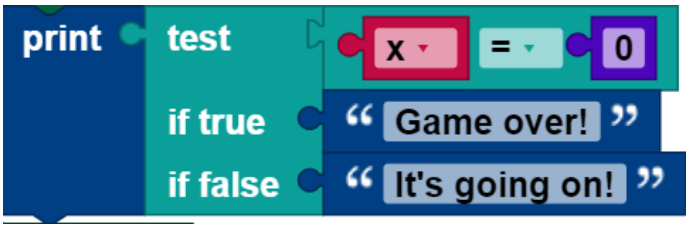
However, leaving an input empty is not recommended.

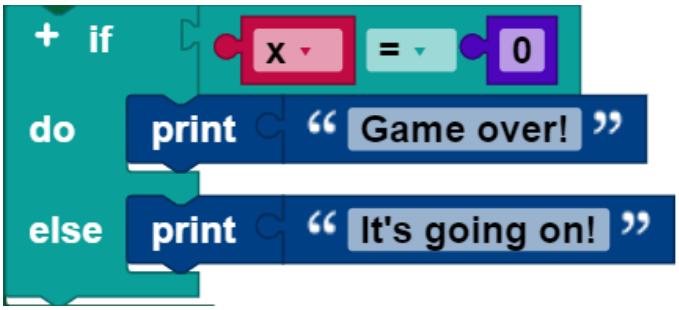
Three-part operator

The three-part operator acts like a miniature **if do** block. It uses three input values The first Boolean condition to be tested is the first input value, the second input value is the value returned if the test is **true**, and the third input value is the value returned if the test is false. In the following example, the variable **color**is set to red if the variable **x** is less than 10, otherwise the variable **color**is set to green.



A three-part block can always be replaced by an **if do** block. The following two examples are just the same.





Loops

The “Controller” area contains blocks that control whether other blocks placed inside them are executed. There are two kinds of control blocks: **if do** blocks (which are described on a separate page) and blocks that control how often the action inside them is executed. The latter are called loops, since the action inside them, called the loop body or body may be repeated multiple times. Each run of a loop is called an iteration.

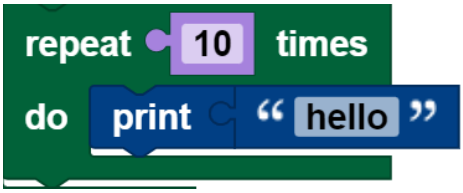
Blocks for creating loops

repeat continuously

The **repeat continuously** block executes the code in the body until the program ends.

repeat

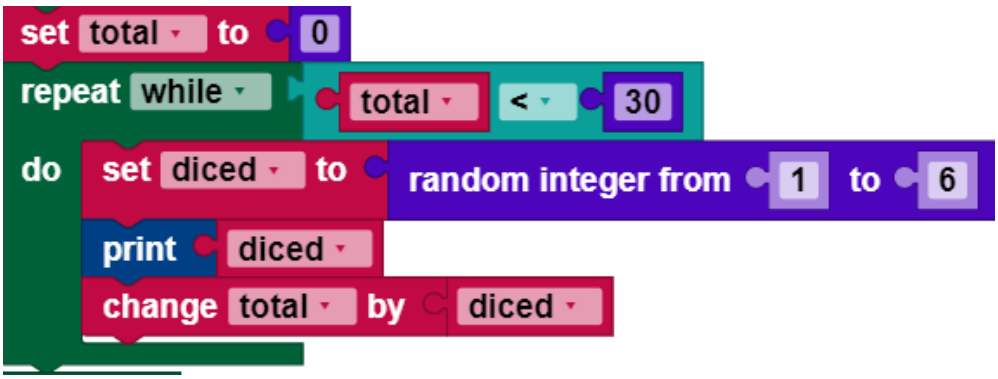
The **repeat** block executes the code in the body as many times ad indicated. The following block, for example, will output “Hello!” ten times:



repeat as long as

Imagine a game in which a player throws a dice and adds up all of the values shown, as long as the total is less than 30. The following blocks implement this game:

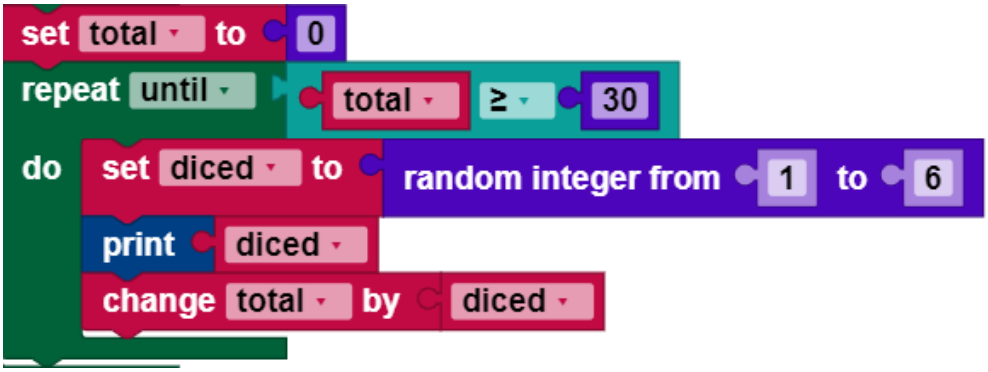
1. A variable named **total** contains an initial value of 0.
2. The loop starts with a check whether **total** is less than 30. If so, the blocks in the body are run.
3. A random integer between 1 and 6 is generated (to simulate a dice value) and a variable named **diced** is saved.
4. The thrown (“diced”) number is output.
5. The variable **total** is increased by the number thrown, or **diced**.
6. Once the end of the loop is reached, the controller goes back to step 2.



After the loop is ended, the controller runs through all of the following blocks (not shown). In the example, the loop ends after a certain number of random integers between 1 and 6 have been output, and the variable **total** then has the value of the total of these numbers, which is at least 30.

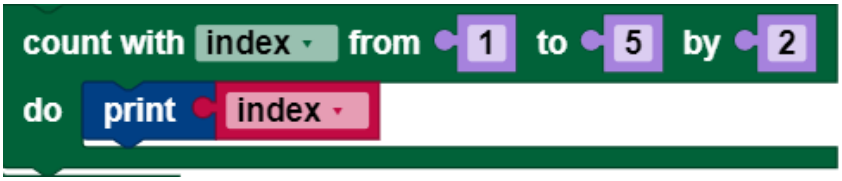
repeat until

repeat as long as loops repeat their body **as long as** a condition is fulfilled. **Repeat until** loops are similar, with the difference that they repeat the body **until** a certain condition is fulfilled. The following blocks are equivalent to the previous example, because the loop runs until **total** is greater than or equal to 30.



count from to

The **count from to** loop increases the value of a variable, starting with an initial value and ending with a second value, and in steps from a third value, whereby the body is executed once for each value of the variable. The following program, for example, outputs the numbers 1, 3, and 5.



As the following two loops show, which each output the numbers 5, 3 and 1, this first value can be greater than the second. The behavior is the same, regardless of whether the incremental amount (third value) is positive or negative.

for each

The **for each** block is similar to the **count from to** loop, but instead of the loop variables in a numerical sequence, it uses the values from a list in sequence. The following program outputs each element in the list “alpha,” “beta,” “gamma”:

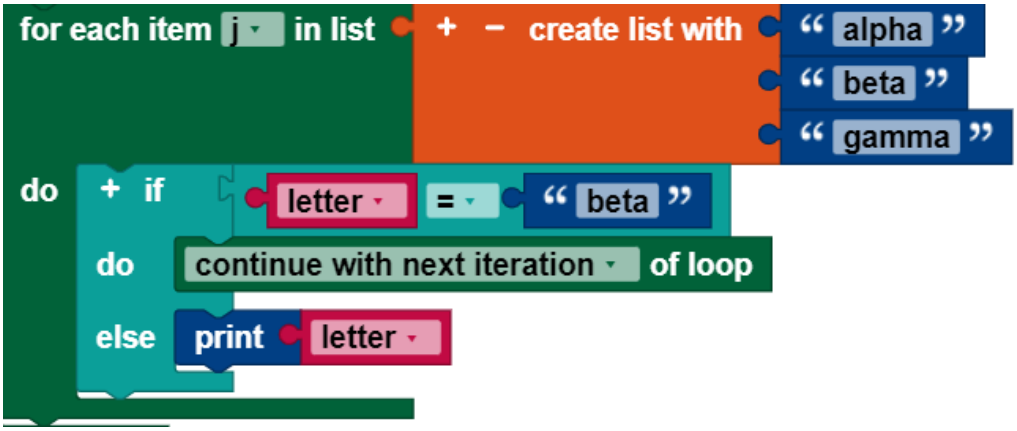
Break out blocks

Most loops are run until the abort condition (for **repeat** blocks) is fulfilled, or until all values for the loop variable have been taken (for **count with** and **for each** loops). Two rarely needed, yet occasionally used blocks offer additional options for controlling loop behavior. They can be used with any kind of loop, even though the following example shows their use in the **for each** loop.

continue with next iteration

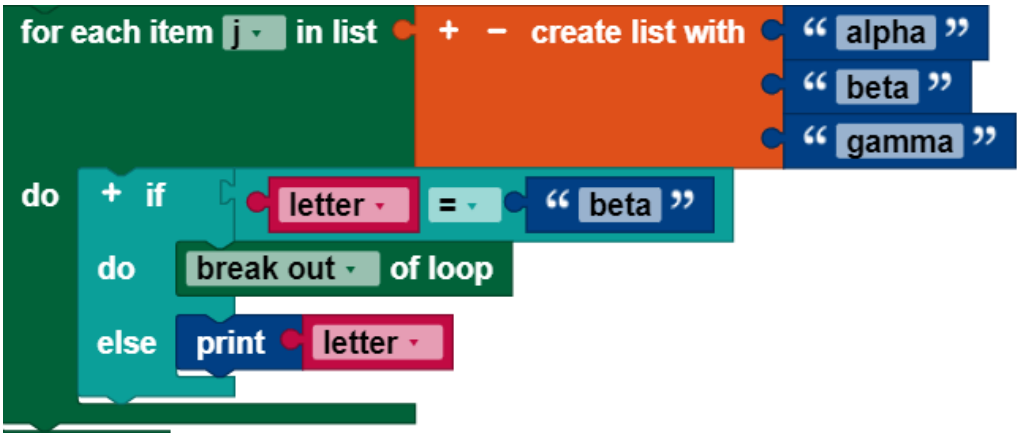
continue with next iteration causes the remaining blocks in the loop body to be skipped, and the next iteration of the loop to begin.

The following program outputs “alpha” during the first iteration of the loop. During the second iteration, the block **continue with next iteration** is executed, causing the output of “beta” to be skipped. In the last iteration, “gamma” is printed.



Break out

The **break out** block makes it possible to prematurely exit a loop. The following program outputs “alpha” for the first iteration, then breaks out of the loop during the second iteration when the loop variable equals “beta.” The third point in the list is never reached.


for each item j in list
+ − create list with
" alpha "
" beta "
" gamma "
do
+ if
letter
=
" beta "
do
break out of loop
else
print
letter

Mathematics

The blocks in the mathematics category are used to activate calculations. The results of the calculations can be used, for example, as values for variables. Most mathematic blocks relate to general mathematical calculations, and should be self-explanatory.

Blocks

Facts and Figures

Use the number block to enter any number into your program, or assign this number to a variable as a value. This program assigns the number 12 to the variable **age** :

Simple calculations

This block has the structure value - operator - value. The available operators are **+**, **-**, **÷**, **×** and **^**. The operator can be selected via the drop down menu. It can be applied directly to numbers, or to values of variables. Example:

This block outputs the result 144 (12^2).

Specialized calculations

This block applies the type of calculation selected from the drop down menu to the number behind it or the variable behind it. The available operations are:

- Square root,
- Sum,
- Natural logarithm,
- Decadic logarithm,
- Exponential functions with the base e (e^1, e^2,...),
- Exponential functions with the base 10 (10^1, 10^2,...),
- Changing sign (multiplying by -1.

e here is Euler's number. This block takes the square root of 16 and sets the variable **i** to the result.

Trigonometric functions

This block works similarly to the block described above, with the difference that the trigonometric functions sine, cosine, tangent, and their inverse functions are used. The number indicated or the value of the variable indicated is therefore inserted into the function selected in the drop down menu, and the result can then be processed in the program. In addition, there is also the block **arctan2 of X: ... Y: ...**, which makes it possible to use two real numbers (to be entered as X and Y) to output a function value for the arctan2 in a range of 360°.

Frequently used constants

This block works similarly to the number block, however you do not enter the numerical value here yourself. Instead, frequently used constants (such as ?) are saved here as defaults. The constants can be selected via the drop down menu.

Remainder of a division

The **remainder of ...** block is used to output the remainder of a division. This program assigns the variable **remainder** to the remainder of the devision of 3:2, or 1:

Round

The **round ...** block can be used to round an entered decimal number of the value of an entered variable to a whole number. You can choose from three options in the drop down menu:

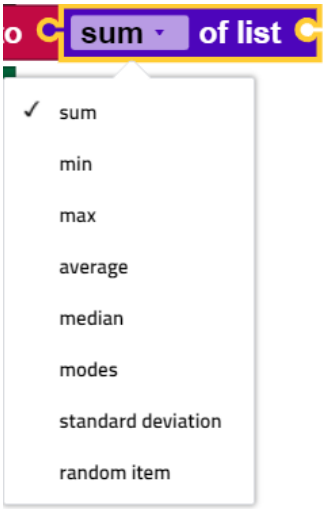
- use “round” for standard rounding (e.g. 4.5 to 5)
- use “round up” for rounding up (e.g. 5.1 to 6)
- use “round down” for rounding down (e.g. 5.9 to 5).

Evaluating lists

You can use the **... of the list** block to output

- the sum of all values in a list with “sum,”
- the smallest value in a list with “min,”
- the largest value in a list with “max,”
- the average of all values in a list with “average,”
- the median of a list with “median,”
- the most frequent value in a list with “mode,”
- the standard deviation of all values in a list with “standard deviation,”
- a random value from a list with “random item”

. You can select all of these options using the drop down menu for the block:



Constrain input values

The **constrain ... from ... to** block allows you to constrain input values to a certain interval. Before an input value is processed, a test is conducted to check whether it is in the defined interval. There are three options for handling an entered value:

- The value is in the interval, so it is transmitted without change.
- The value is below the lower limit for the interval, so this lower limit is transmitted.
- The value is above the upper limit for the interval, so this upper limit is transmitted.

In this example, the block is used to constrain the value for the variable **speed** to the speeds supported by the motor:

Generate random values

The two blocks **random number from ... to...** and **random break** output a random value. The **random number from ... to... block** outputs a number from the defined interval. The block **random break**, in contrast, outputs a value between 0.0 (and may include this number) and 1.0 (may not include this number).

Text

Texts Examples of texts are:

“Thing 1”
“12. March 2010”
“” (empty text)
Text can contain letters (capital or lower case), numbers, punctuation marks, other symbols, and spaces.

Blocks

Creating text

The following block creates the text “Hello” and saves it in the variable named **greeting:**

The block **create text with** combines the value of the variable **greeting** and the new text “world” to create the text “Helloworld.” Please note that there is no space between the two texts, since there was none in the original texts.

To increase the number of text inputs, click the (+) symbol. To remove the last output, click the (-) symbol.

Changing text

The block **to ... append** adds the entered text to the given variable. In this example, it changes the value for the variable **greeting** from “Hello” to “Hello, there!”:

Text length

The **length of** block counts the number of characters (letters, numbers, etc.) contained in a text. The length of “We are #1!” is 12, and the length of the empty text is 0.

Check for empty text

This **is empty** block checks whether the entered text is empty (the length is 0). The result is **true** in the first example, and **false** in the second example.

Search for text

These blocks can be used to check whether a text is present in another text, and if so, where. For example, this block checks for the first occurrence of “e” in “Hello,” and the result is 2:

This one checks for the last occurrence of “e” in Hello, which is also 2:

Regardless of whether the first or last occurrence is selected, this block delivers the result 0, since “Hello” does not contain a “z.”

Extracting text

Extracting a single character

This returns “b,” the second letter in “abcde”:

This returns “d,” the next to last letter in “abcde”:

This returns “a,” the first letter in “abcde”:

This returns “e,” the last letter in “abcde”:

This contains each of the 5 letters in “abcde” with the same probability:

None of them changes the text from which these results are extracted.

Extracting a text range

The **in text ... get substring from** block can be used to extract a text range that either starts with:

- letter #
- letter # from end
- first letter

and ends with:

- letter #
- letter # from end
- last letter

In the following example, “abc” is extracted:

Change text capitalization

This block generates a version of the input text either written in

- UPPER CASE (all letters in caps) or
- lower case (all letters as lower case), or
- Substantive (first letters capitalized, other letters lower case).

The result of the following block is “HELLO”:

Non-alphabetic characters are not affected. Please note that this block does not work on text in languages without capital and lower case letters, like Chinese.

Trimming (removing) spaces

The following block removes spaces, depending on the settings in the drop down menu (small triangle):

- at the start of the text
- at the end of the text
- on both sides of the text

The result of the following block is “Hi you.”

Spaces in the middle of the text are not affected.

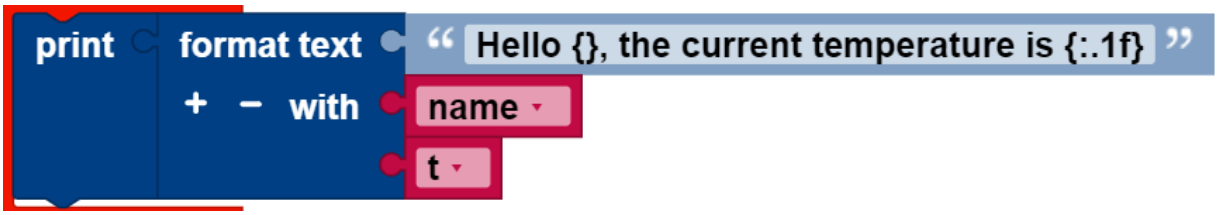
Print text

The **print** block causes the input value in the console window to be printed:

It is never sent to the printer, although the name might seem to indicate this.

Output text with formatting

You can use the **formatted text** block to output texts with formatted variable content. All place holders **{}** in the text are replaced with the content of the variables appended after the text. Formatting can be entered into the brackets. The formatting **{:.1f}**, for instance, outputs only the first decimal place in the variable **t**.



Data structures

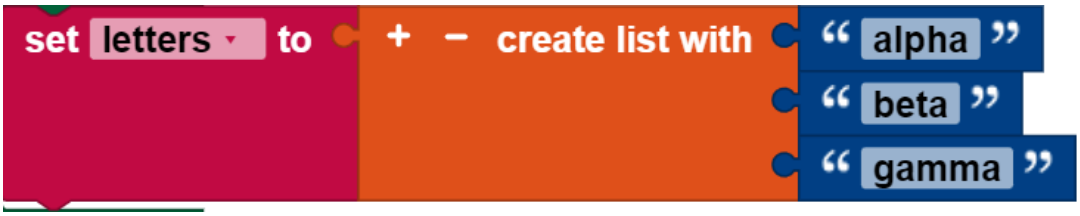
Lists

As in everyday language, in ROBO Pro Coding a list is an ordered collection of elements, such as a “to do” list or a shopping list. Elements in a list can be of any type, and the same value can appear in a list multiple times.

Creating a list

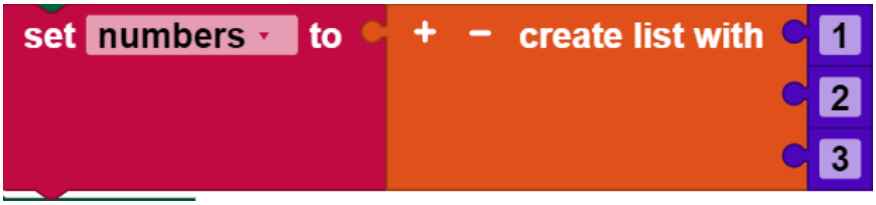
create list with

You can use the **create list with** block to enter the initial values in a new list. In this example, a list of words is created and saved in a variable named **letters**:



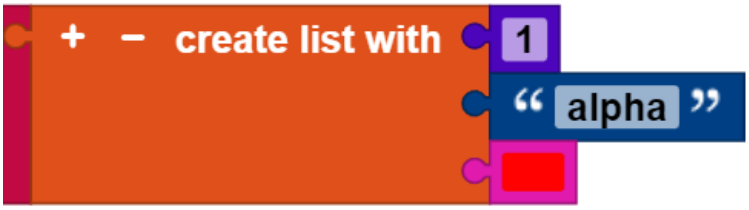
We designate this list as [“alpha,” “beta,” “gamma”].

This shows the block for creating a list of **numbers**:



Here is how to create a list of **colors**:

It is less common, but possible to create a list of values of different types:



Change number of inputs

To change the number of inputs, click or touch the gear symbol. This will open a new window. You can drag element sub-blocks from the left side of the window to the list block on the right side to add a new input:

While the new element in this example is inserted at the bottom, it can be added anywhere. Similarly, element sub-blocks that are not desired can be dragged to the left and out of the list block.

Create list with item

You can use the **create list with item** block to create a list containing the indicated number of copies of an item. The following blocks, for example, set the variable **words** on the list [“very,” “very,” “very”].

Check the length of a list

is empty

The value of an **is empty** block is **true** if its input is the empty list, and **false** if it is anything else. Is this input **true** ? The value of the following block would be **false**, because the variable color is not empty: It has three items.

Note how similar this is to the **is empty** block for text.

Length of

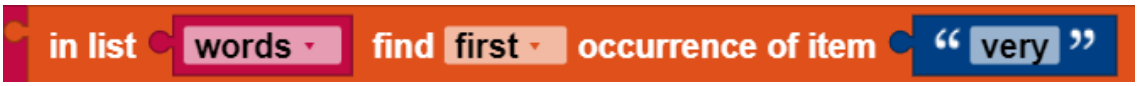
The value of the **length of** block is the number of elements that are in the list used as the input. The value of the following block would be 3, for instance, since **color** has three elements:

The value of the **length of** block is the number of items in the list used as the input. The value of the following block would be 3, for example, although **words** consists of three copies of the same text:

Note how similar this is to the block **length of** for the text.

Searching for items in a list

These blocks find the position of an item in a list. The following example has a value of 1, because the first occurrence of “very” is at the start of the list of words ([“very,” “very,” “very”]).



The result of the following is 3, because the last occurrence of “very” in the **words** is at position 3.

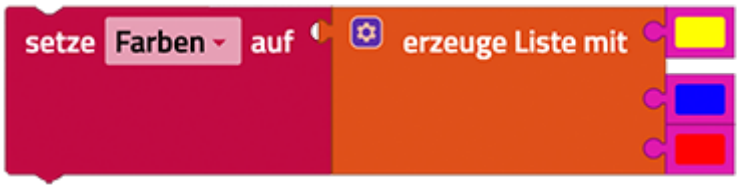
If the item is not in the list at all, then the result is a value of 0, as in this example:

These blocks behave the same way as the blocks for finding letters in text.

Getting items from a list

Getting a single element

Remember the definition of the list **colors**:



The following block contains the color blue, because it is the second item in the list (starting from the left):

This one contains green, because it is the second element (starting from the right end):

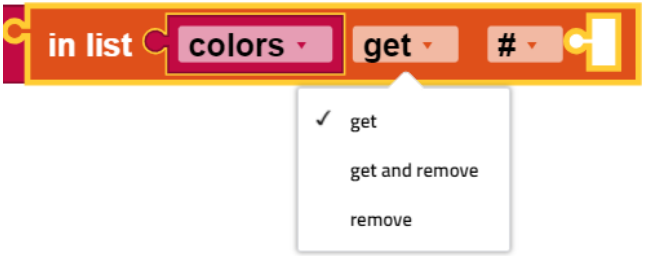
This contains the first item, red:

This contains the last item, yellow:

This one chooses a random item from the list, with the same probability of returning one of the items red, blue, green or yellow.

Get and remove an item

You can use the drop down menu to change the block **in list ... get** to the block **in list ... get and remove**, which delivers the same output, but also changes the list:



this example sets the variable **first letter** to “alpha” and leaves the remaining letters ([“beta,” “gamma”]) in the list.

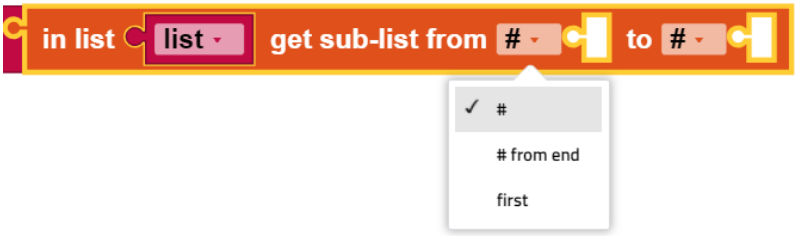
Removing an entry

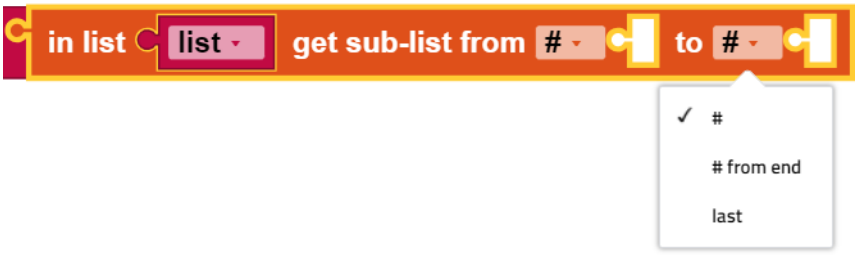
If you select **remove** from the drop down menu, the tab at the left of the block will be removed:

Then, the first item from **letter** will be removed.

Get a sub-list

The block **in list ... get sub-list** is similar to the block **in list ... get**, with the difference that it extracts a sub-list and not an individual item. There are multiple options to enter the start and end of the sub-list:





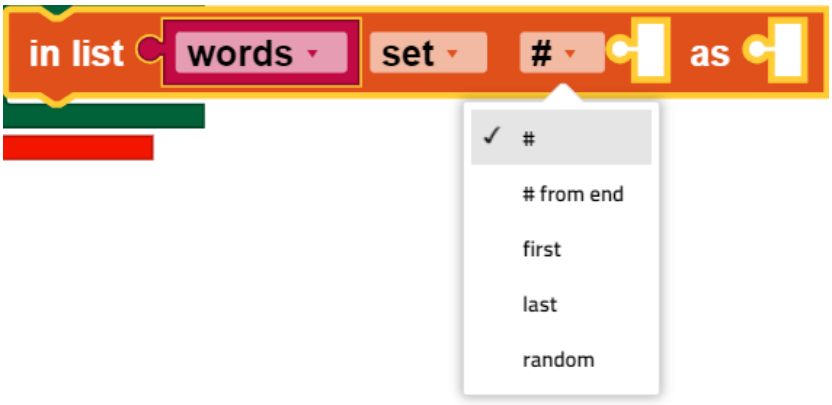
In this example, a new list **first letters** is created. The new list has two items: ["alpha," "beta"].

Please note that this block does not change the original list.

Adding items to a list

Replacing items in a list

The block **in list ... set** replaces the item at a certain point in a list with another item.



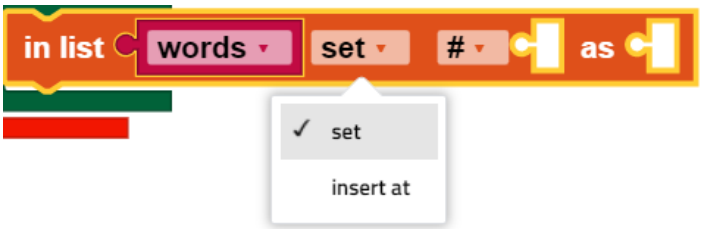
The meanings of the individual drop down options are outlined in the previous section.

The following example does two things:

1. The list **words** is created with 3 items: [“very,” “very,” “very”].
2. The third item in the list is replaced with “good.” The new value of **words** is [“very,” “very,” “good”]

Insert items from a certain point into a list

The **in list ... insert at** block is accessed via the drop down menu for the **in list ... set** block:



It inserts a new item at the indicated point into the list, before the element that was previously located there. The following example (which builds on an earlier example) does three things:

1. The list **words** is created with 3 items: [“very,” “very,” “very”].
2. The third item in the list is replaced with “good.” The new value of **words** is therefore [“very,” “very,” “good”].
3. The word “Be” is inserted at the start of the list. The final value of **words** is therefore [“Be,” “very,” “very,” “good”].

Divide character strings and merge lists

Make list from text

The block **make list from text** uses a delimiter to divide the given text into parts:

In the example above, a new list will be returned containing three segments of text: "311," "555" and "2368".

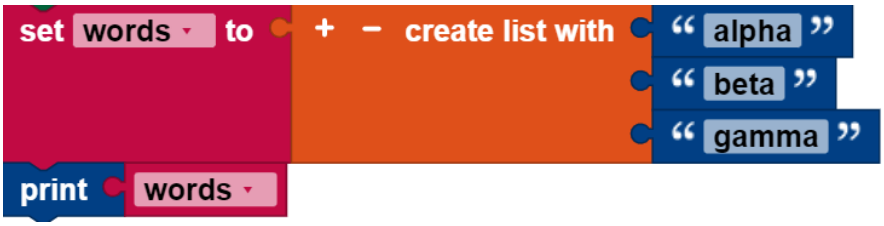
Make text from list

This block **make text from list** assembles a list into a single text using a delimiter:

Related blocks

Printing a list

The **print** block in the text category can output lists. The result of the following program is the console output shown:



Console

Program starts ...

['alpha', 'beta', 'gamma']

Program finished.

Complete something for each element in a list

The **for each** block in the controller category executes an operation for each element in a list. This block, for example, prints each item in the list individually:

The items in this case are not removed from the original list.

See also the examples for break out blocks.

Map

JSON

Util

The usage category contains blocks of the following type in ROBO Pro Coding:

- Color selection
- Wait
- Python Code
- Start
- Function execution

Color selection

This block serves as an input value if a color is queries (for instance if the camera is completing a color comparison). You can click or touch the color to choose one of a range of 70 colors.

Wait

Wait until the time has expired

The block **wait [] ...** prevents the program from continuing to run for the indicated wait time. You can select the time unit in the drop down menu (small triangle) as well as the desired pause length in the input field beside it.

Wait with condition

In the **wait until** block, the pause is linked not to the time but to the fulfillment of a condition (such as whether a button is pressed). The condition is added to the **wait until** block.

Python Code

If you would like to integrate existing Python Code into ROBO Pro Coding, you can insert it into the **Python Code** block. The program will then execute everything written in the block in Python.

Start

The **start when** block is also linked to a condition. The program in the block body will only start once this condition is fulfilled.

Function execution

You can use the **execute function ...** in a thread to execute the selected function in a separate thread. In some cases, this measure can allow the program to continue reacting to inputs and to be executed more quickly.

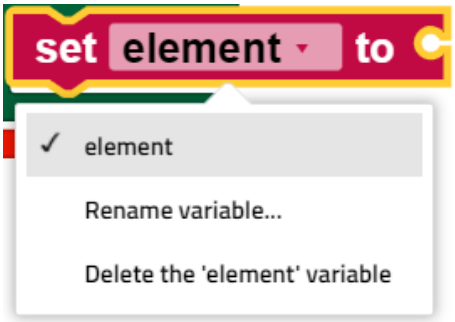
Variables

We use the term variable as it is used in mathematics and other programming languages: a named value that can be changed (varied). Variables can be created in different ways.

- Some blocks like **count with** and **for each** use a variable and define its values. A traditional IT term for such variables is loop variables.
- User-defined functions (also called “procedures” can define inputs, which can be used to create variables that can only be used in this function. Such variables are traditionally referred to as “parameters” or “arguments.”
- Users can change variables at any time using the **set** block. These are traditionally called “global variables.” They can be used anywhere in the code of ROBO Pro Coding.

Drop down menu

When you click the drop down symbol (small triangle) for a variable, the following menu appears:



The menu offers the following options.

- display the names of all available variables defined in the program.
- “rename variable ...,” e.g. change the name of this variable wherever it appears in the program (choosing this option will open a query asking for the new name)
- “delete variable ...,” e.g. Delete all blocks that refer to this variable, wherever they are in the program.

Blocks

Set

The **set** block assigns a value to a variable, and creates the variable if it does not yet exist. For example, this is how to set the value for the variable **age** to 12:

Call

The **call** block delivers a variable saved in a variable without changing it:

It is possible to write a program containing a **call** block without a relevant set block, but this is a bad idea.

Change

The **change** block inserts a number for a variable.

The **change** block is an abbreviation for the following construct:

Example

Look at the following example code:

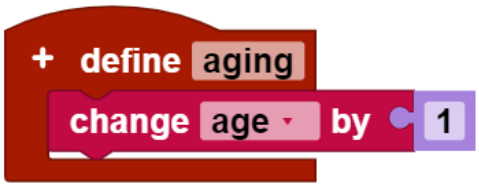
The first row of blocks creates a variable named **age** and **sets** its initial value to the number 12. The second row of blocks **calls** up the value 12, adds 1 to it, and saves the total (13) in the variables. In the last line, the following message appears: “Congratulations! You are now 13.”

Functions

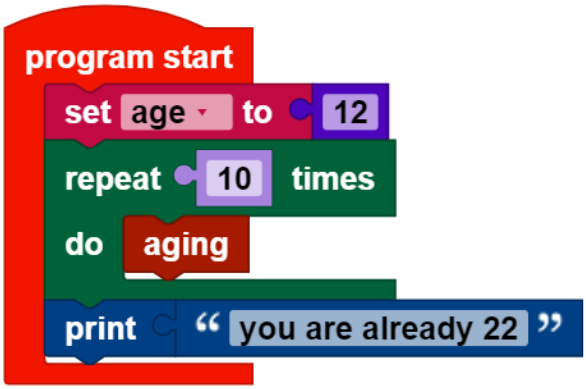
Functions are used to make parts of the code reusable and thereby provide an overarching structure for the code. If you complete a function block, a new block will appear in the Functions menu with the same name as this function block. Now, it is possible to simply insert the block with the name of the function in the main program. When the program is run, this block will lead to the code in the function of the same name, and process this code.

Simple function

The simple function block can be used to create a function bearing the name entered in the text field. This function can contain as many variables as desired, which are added using the gear symbol. This function **aging** adds 1 to the variable **age**:

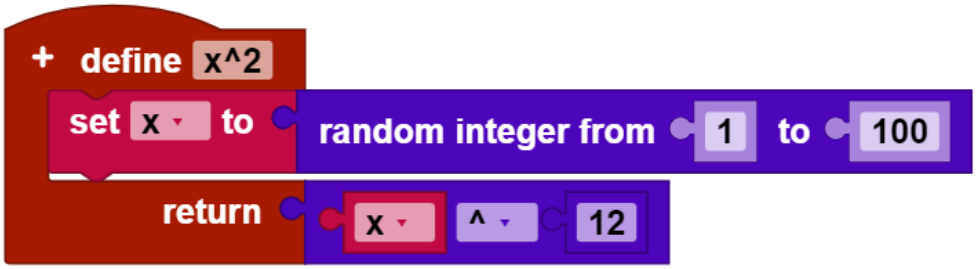


The function can then be used in the main program:



Function with return value

This block makes it possible to create a function with return value. The return value can then be used in the main program. Here is an example:



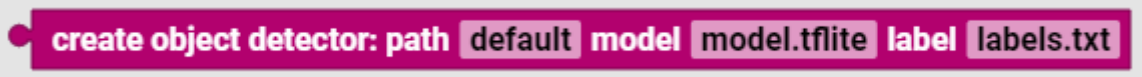
Machine Learning

The group "Machine Learning" contains blocks for using with TensorFlow project and the USB camera.

Creates an image analysis for a model in the path on the TXT 4.0 controller.

Creates an object recognition with a standard model "sorting route with AI".

Creates an object identifier for a model in the path on the TXT 4.0 controller.



Analyses an image with object or image recognition. The recognised properties, their probability and, in the case of object recognition, their position are output. The result of this block can be written into a variable in order to evaluate it later in the "get value of result item [item]" block. block.

Outputs a single value of a property of the xth result of an image or object analysis. The item can either be the "process image" block directly or its results from a variable.

Imports

Imports contains all functions from self-defined modules in "lib".

Functions are used to make parts of the code reusable and thus to structure the code as a whole.

see "Functions"