
JobBlockly Documentation

Celine Deknop

juin 25, 2018

Table des matières

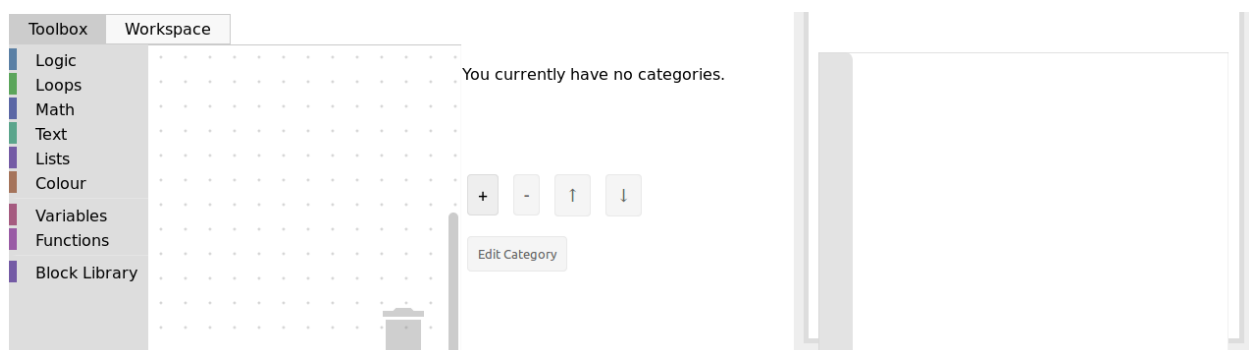
1	Comment créer une tâche Blockly <i>basique</i> ?	1
1.1	Example : create the sum function (using the graphical interface)	2
1.2	Example : create the sum function by hand	11
1.3	Example : an « only workspace » task	14
1.4	Example : create a custom block (if/else)	21
2	Comment créer une tâche labyrinthe ?	29
3	Comment changer le visuel du labyrinthe	33
3.1	Files you will need	33
3.2	Files to modify	35
4	Indices et tables	37

Comment créer une tâche Blockly *basique* ?

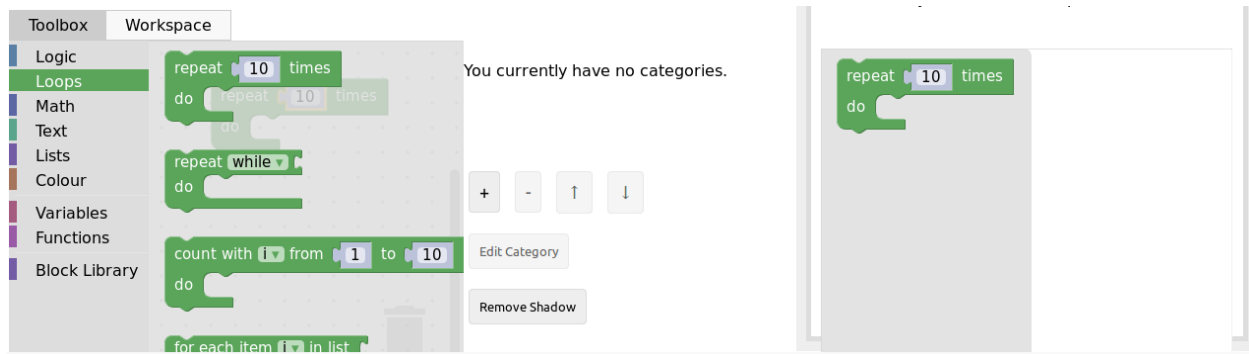
There is, first, a few steps that are the same as for any other tasks. Those are :

1. Create the exercise as you would a classical one. Set up a title, a context, your name, the options you want, ...
When creating a subproblem, select « blockly » as « type of task »
2. Again, perform the set-up of the task as you normally would
3. If you want, set up the maximum number of blocks that the student can use to perform the task by entering it in the « Max number of blocks » field (by default, it is « Infinity »)

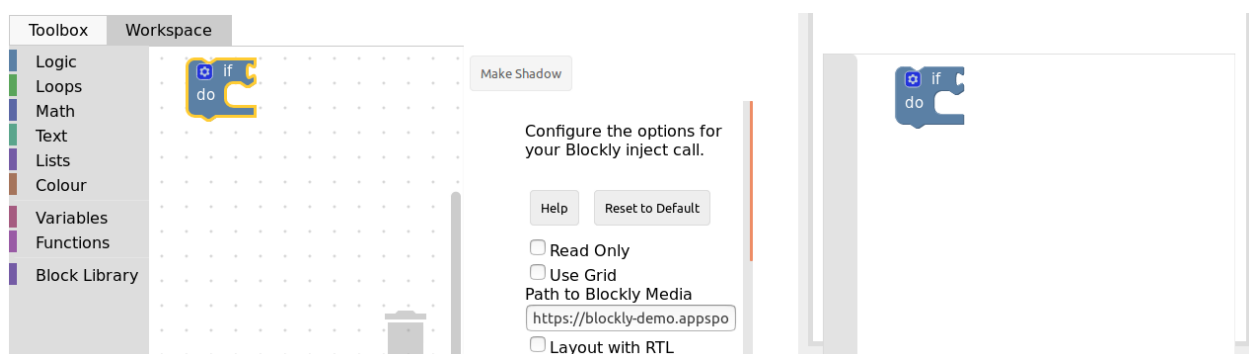
Now, there is two ways to configure Blockly : either using the embedded graphical interface or by entering the blocks by hand. Since the first solution is more beginner-friendly, let's explore it first. Scroll down all the way and click « edit toolbox/workspace graphically ». This is what you will see.



The left side is where you can configure the tool, and the right side will display a live preview of what you did so far. The left side has two tabs : the toolbox will hold the pool of blocks that the student can use to solve the task. To add blocks, simply click on one category and drag/drop the block you want in the tab. Here is an example :



If you want to delete a block, simply drag it to the trashcan on the bottom right. Now, you can also add blocks to the workspace of the student, that will serve as a base for the exercise. Simply click on the « workspace » tab and drag/drop the same way.

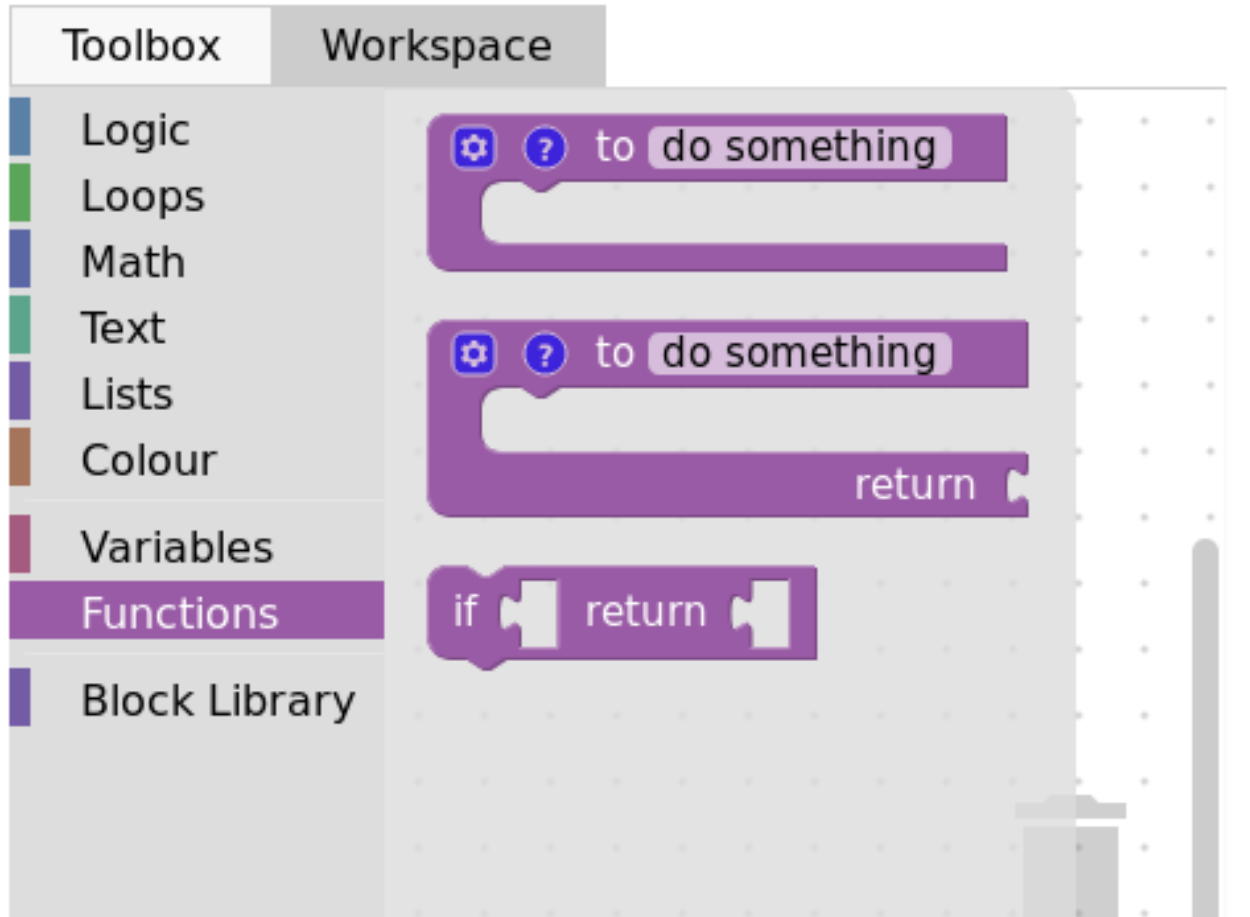


Let's now see an example of what can be done for a simple exercise.

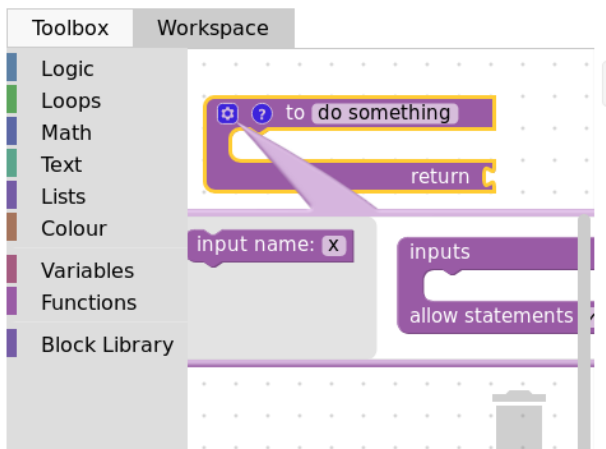
1.1 Example : create the sum function (using the graphical interface)

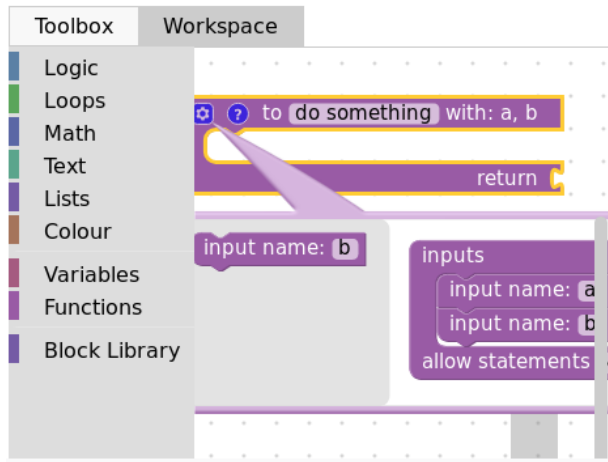
Here, we are in the case where we want the student to create a function, which means we have to provide him with its signature in the workspace. Our Sum function needs to take in two parameters, the two numbers to sum (let's call them a and b), and return the resulting sum.

First, click the « Workspace » tab and open the « Function » category. Out of the three blocks, we need the functions that returns, which is the second block on the image here.

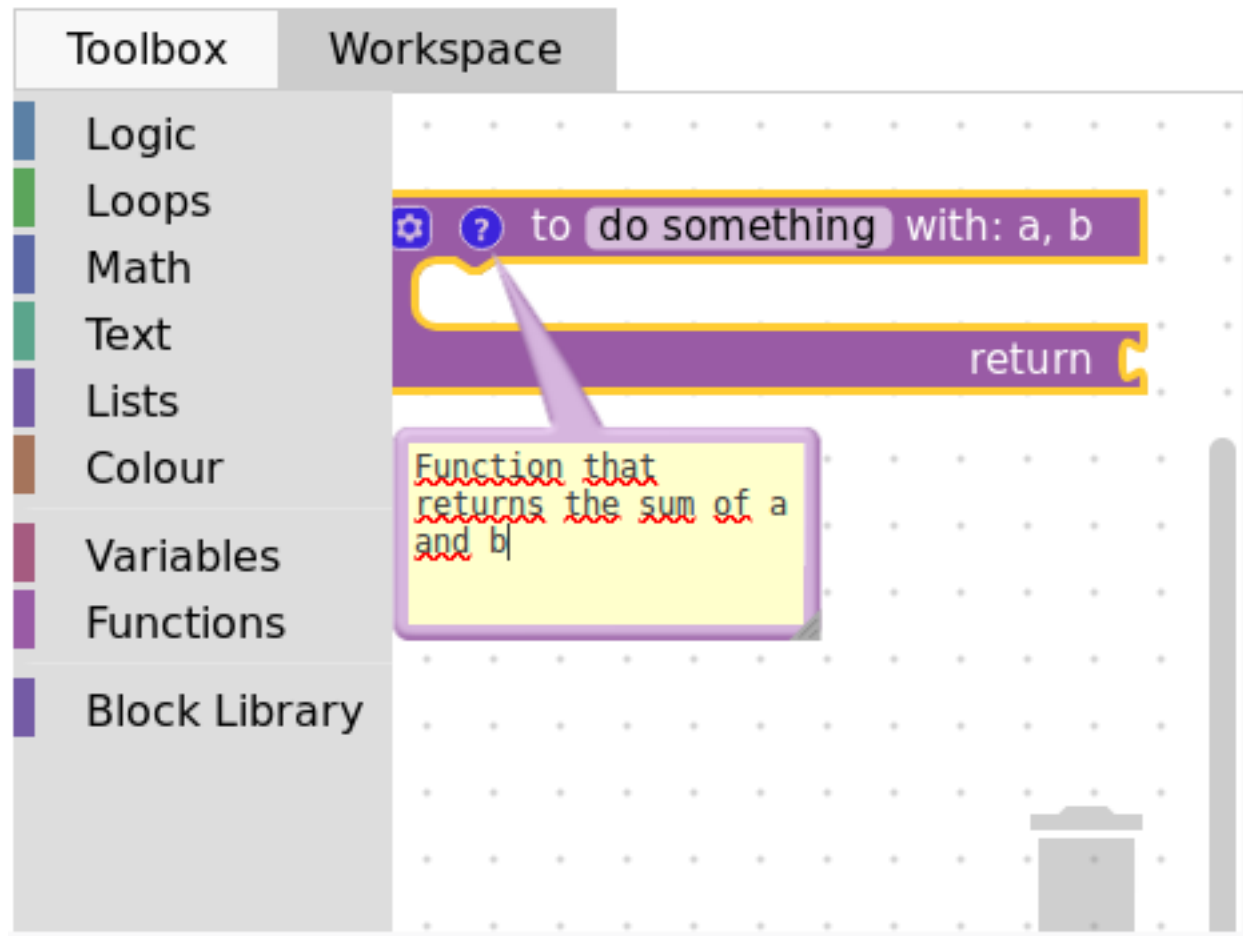


Now, configure the function. The wheel icons allow us to add parameters. Simply name your parameter (x by default), then connect the block into the right space, like so :

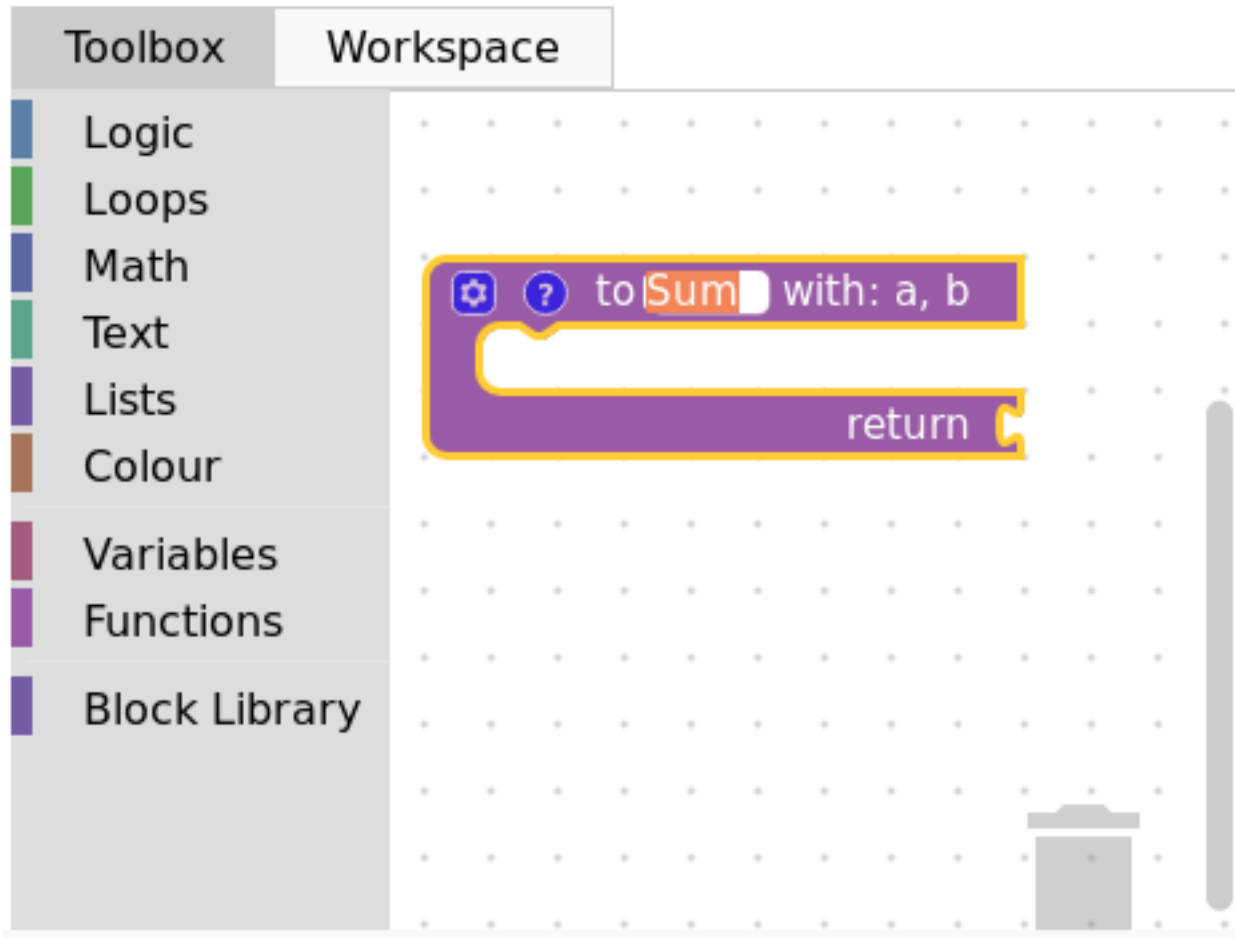




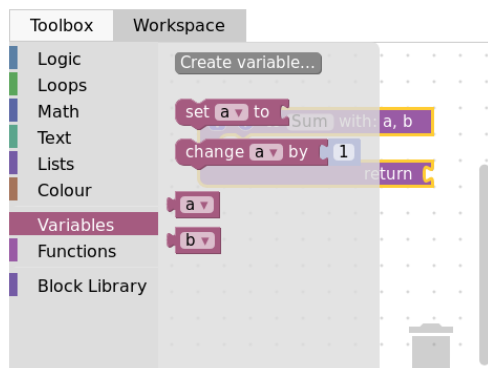
The ? icon allows us to set a tooltip (text that show on mouseover) simply by typing in the field :



Finally, we have to name our function, changing the *do something* into what we want, here, *Sum* :

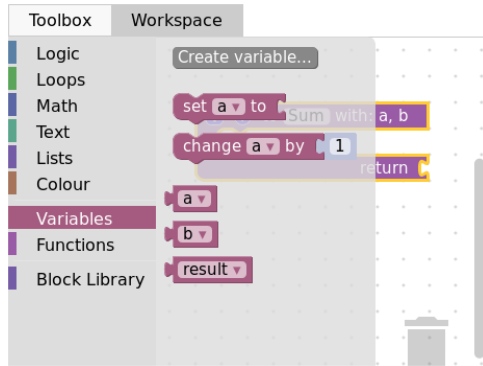


Now, let's create a variable to hold the result. Click on the « Variables » category and select « create variable ». Input your variable name, « result » for example, and it will be available in the category :

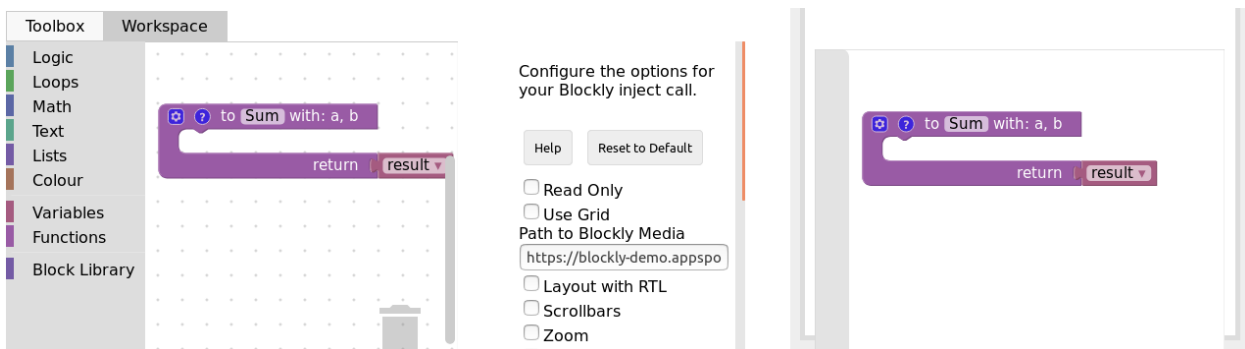


New variable name:

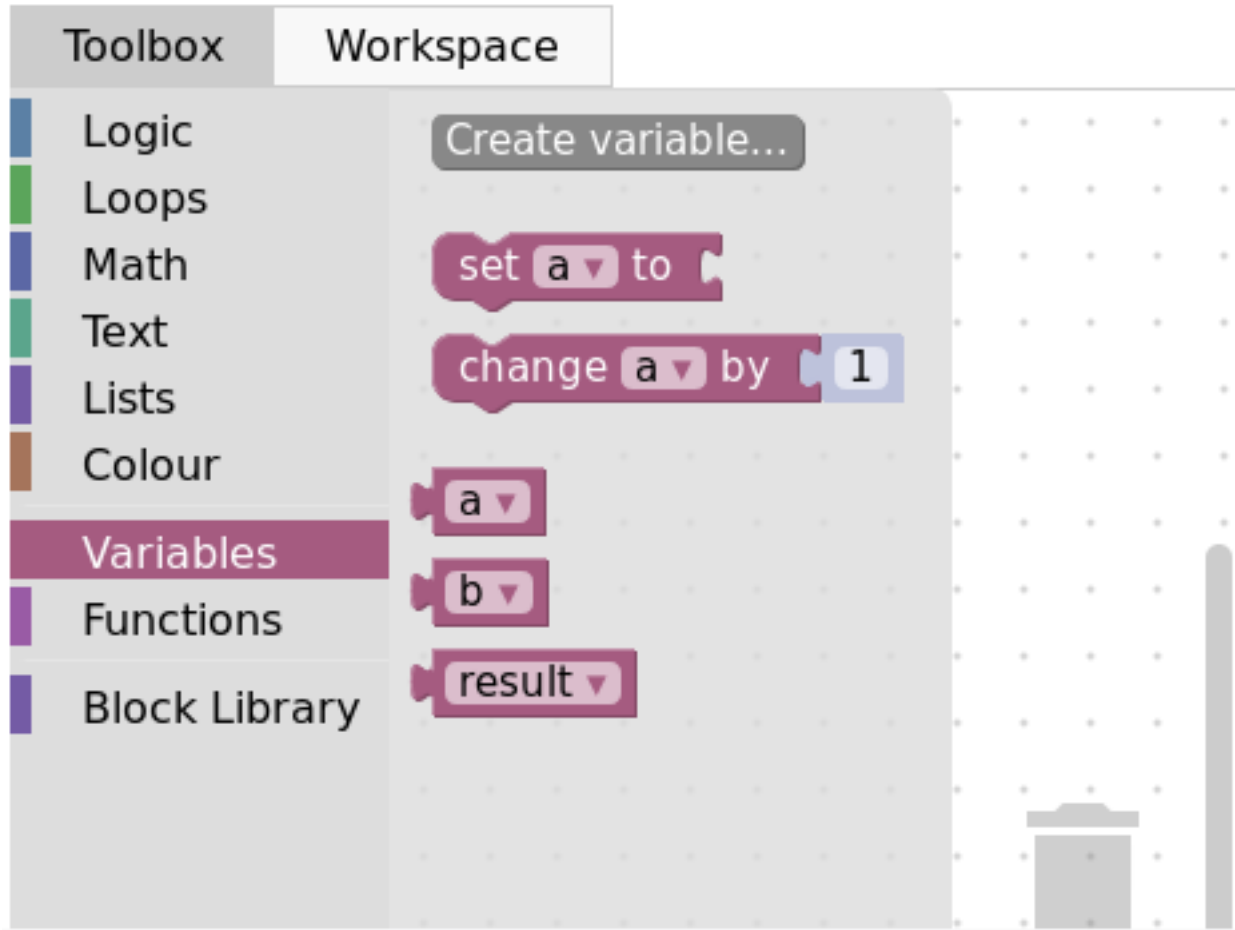
Annuler OK



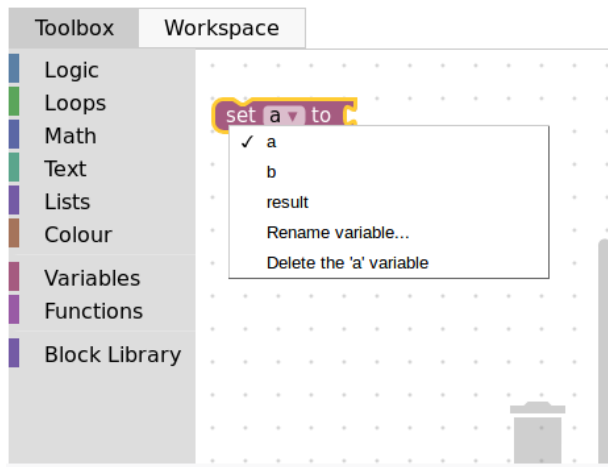
Finally, select the corresponding block and plug it into the « return » spot. Here is our basic workspace done, with the preview :

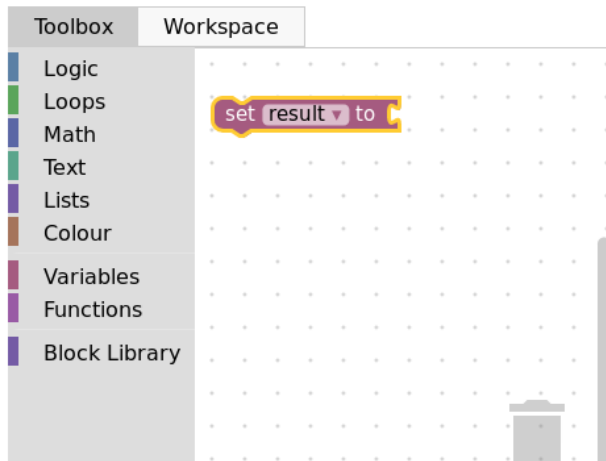


Now, it is time to create the toolbox. Click on the corresponding tab, and select the blocks that you want for the task. In our case, we first need to re-create all the previous variables, the same way as we did for the *result* one (clicking on create variable). Here is what we end up with :

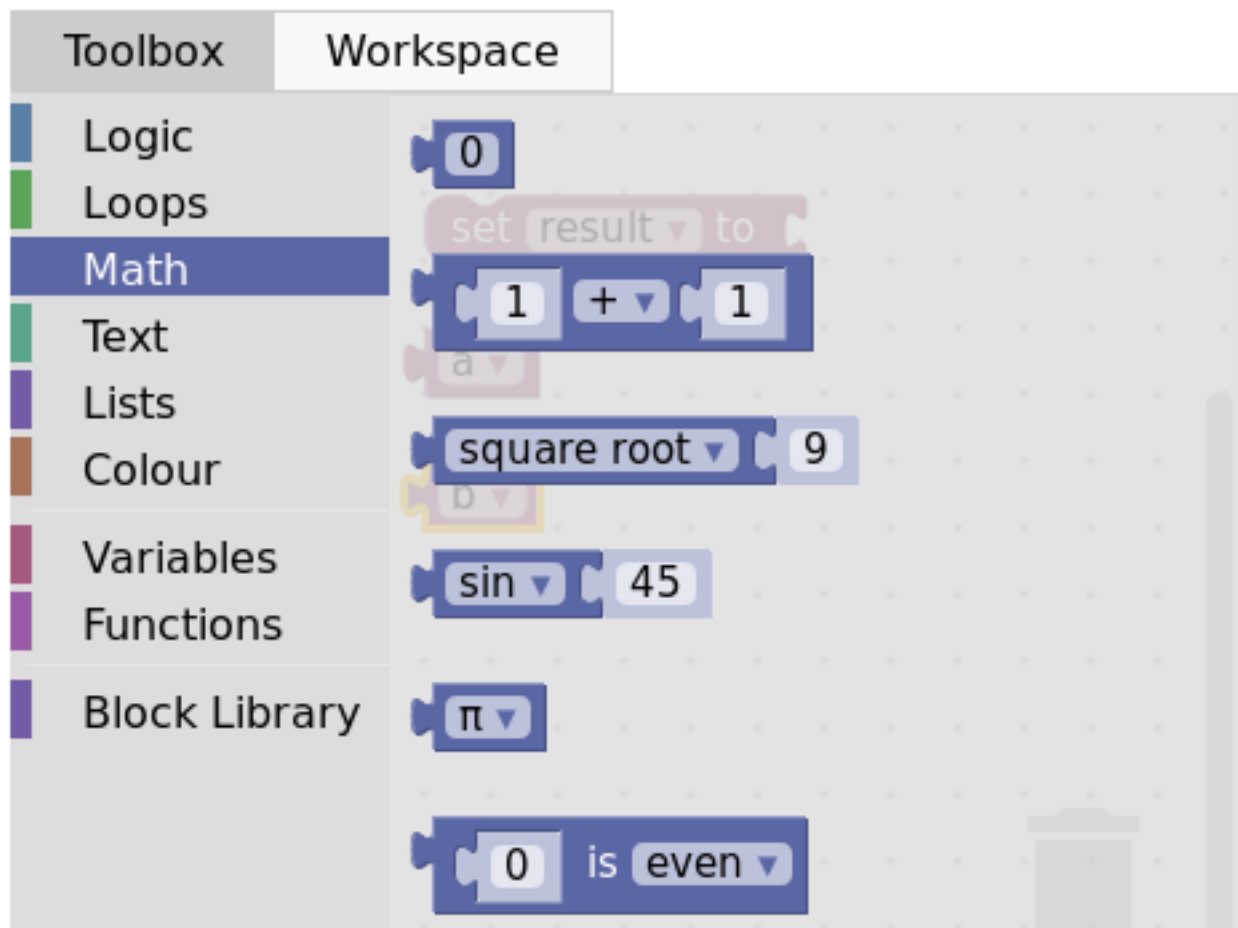


Then, we want the *set* block, so we drag it to the toolbox. Using the arrow next to the variable name, we can select the variable we want by default (*result* in our case) :

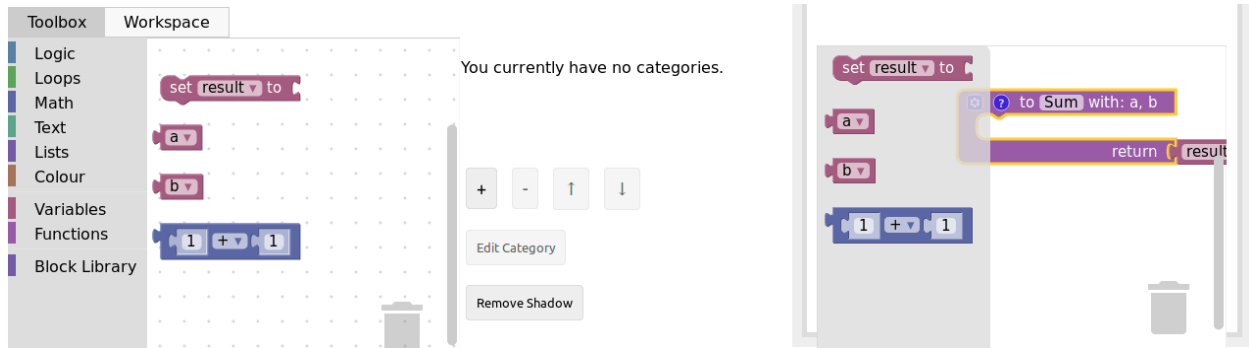




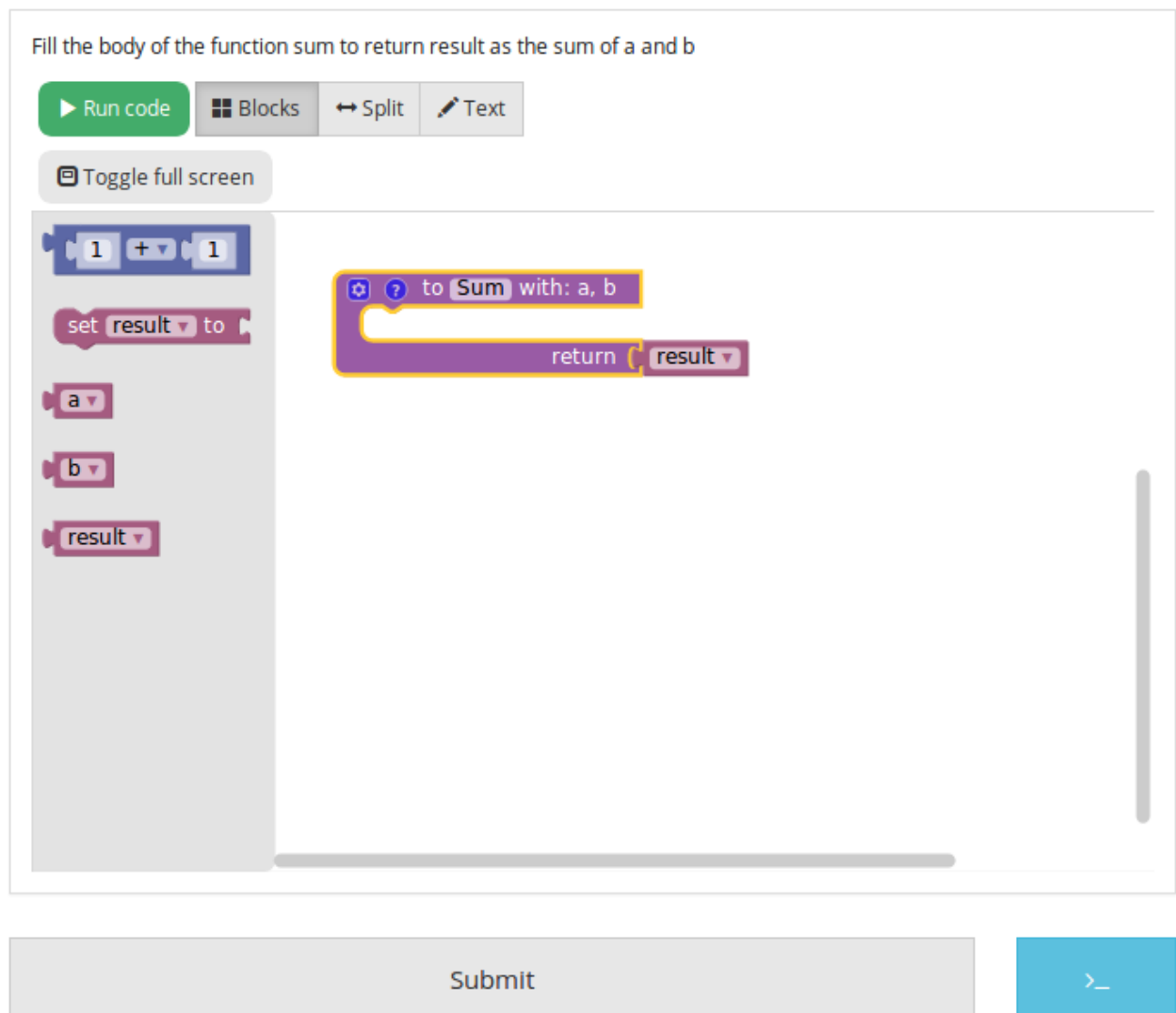
Then, we add the two previously created variables « a » and « b » as well. Finally, we want the sum operator from the math category :



And here is the final product with the preview :



Click close, then save, and you are done with the graphical interface part of the task creation. You can now visualize your task on INGINious and connect blocks, but there is no correction or feedback yet. Here is what it will look like to the student :



For the feedback, you'll have to create a run and a file that contains the task correction. Let's start with that one, that we will call `sum.py`. It has to first get the student's code with an instruction like this : `@@subProblemID@@`. Then, you will be able to call the created function with it's name (here « Sum »), and then run any tests you want. To comply with the usual INGINious run file, you have to output « True » if the tests pass, and some feedback followed

by `exit()` for a failure. The following code is an example for our sum function :

```
#!/bin/python3
#Open source licence goes here

from contextlib import redirect_stdout
import random

@@Sum@@ #The id of your subproblem goes here

if __name__ == "__main__":
    random.seed(55)
    for j in range(6): #let's test 6 times
        a = random.randint(0,10)
        b = random.randint(0,10)
        result = Sum(a, b)
        if(result != (a+b)):
            print("The sum you returned for the values " + str(a) + " and " + str(b) +
                  " is " + str(result) + " when the correct answer is " + str(a+b) + ".")
            exit()
    print("True")
```

For such a simple task, the basic run file is sufficient, with only two lines to modify, where you will have to put the name of your correction file. Here is the corresponding code for our sum task :

```
#!/bin/python3
#Open source licence goes here

import os
import subprocess
import shlex
from inginius import feedback
from inginius import input

if __name__ == "__main__":
    input.parse_template("sum.py") #Replace sum.py by your filename on this line and
    ↳the next
    p = subprocess.Popen(shlex.split("python3 sum.py"), stderr=subprocess.STDOUT,
    ↳stdout=subprocess.PIPE)
    make_output = p.communicate()[0].decode('utf-8')
    if p.returncode:
        feedback.set_global_result("failed")
        feedback.set_global_feedback("Your code could not be executed. Please verify
    ↳that all your blocks are correctly connected.")
        exit(0)
    elif make_output == "True\n":
        feedback.set_global_result("success")
        feedback.set_global_feedback("You solved the task !")
    else:
        feedback.set_global_result("failed")
        feedback.set_global_feedback("You made a mistake ! " + make_output)
```

Those two files need to go in your task folder, and the task creation is complete !

1.2 Example : create the sum function by hand

Both the toolbox and the workspace can also be created by hand (using xml code) when clicking on the « Edit toolbox XML » and « Edit workspace XML » buttons. We'll go over how to configure those two to achieve the same set up as the previous example.

First, xml tags must surround every other lines in both the toolbox and the workspace, like this :

```
<xml xmlns="http://www.w3.org/1999/xhtml">
</xml>
```

Then, for the toolbox, we need the variables *a*, *b* and *result*. The code for one variable is the following, only the content of the `field` tag changes to indicate the variable name. Here is the code for variable *a* :

```
<block type="variables_get">
  <field name="VAR">a</field>
</block>
```

We also need the sum operator block code, which is the following :

```
<block type="math_arithmetic">
  <field name="OP">ADD</field>
  <value name="A">
    <shadow type="math_number">
      <field name="NUM">1</field>
    </shadow>
  </value>
  <value name="B">
    <shadow type="math_number">
      <field name="NUM">1</field>
    </shadow>
  </value>
</block>
```

Each block will have different code, that you can find either online or by using the graphical interface. You can also customize a block by modifying the values (changing *ADD* for *MINUS* in the `field` tag will give you a minus operator block, for example).

To recapitulate, this is the full code for the toolbox :

```
<xml xmlns="http://www.w3.org/1999/xhtml">
  <block type="math_arithmetic">
    <field name="OP">ADD</field>
    <value name="A">
      <shadow type="math_number">
        <field name="NUM">1</field>
      </shadow>
    </value>
    <value name="B">
      <shadow type="math_number">
        <field name="NUM">1</field>
      </shadow>
    </value>
  </block>
  <block type="variables_set">
    <field name="VAR">result</field>
  </block>
```

(suite sur la page suivante)

(suite de la page précédente)

```

<block type="variables_get">
  <field name="VAR">a</field>
</block>
<block type="variables_get">
  <field name="VAR">b</field>
</block>
<block type="variables_get">
  <field name="VAR">result</field>
</block>
</xml>

```

Now, for the workspace, we need our function again. The arguments are specified in the `mutation` tag, the name under `name` and the tooltip under `comment`. Finally, our result variable is specified by a special `value` tag, with the name *RETURN*. Here is the code for the workspace.

```

<xml xmlns="http://www.w3.org/1999/xhtml">
  <block type="procedures_defreturn" deletable="false">
    <mutation>
      <arg name="a"></arg>
      <arg name="b"></arg>
    </mutation>
    <field name="NAME">Sum</field>
    <comment pinned="false" h="80" w="160">Return the sum of values a and b...</comment>
    <value name="RETURN">
      <block type="variables_get">
        <field name="VAR">result</field>
      </block>
    </value>
  </block>
</xml>

```

At this point, we have the exact same result as in the previous example. But modifying the toolbox by hand might give you a finer control over the final display. For example, we could create a *Variable* and a *Math* category, which will make the display lighter. This can be done with `category` tags, like so :

```

<xml xmlns="http://www.w3.org/1999/xhtml">
  <category name="Math">
    <block type="math_arithmetic">
      <field name="OP">ADD</field>
      <value name="A">
        <shadow type="math_number">
          <field name="NUM">1</field>
        </shadow>
      </value>
      <value name="B">
        <shadow type="math_number">
          <field name="NUM">1</field>
        </shadow>
      </value>
    </block>
  </category>
  <category name="Variables">
    <block type="variables_set">
      <field name="VAR">result</field>
    </block>
  </category>
</xml>

```

(suite sur la page suivante)

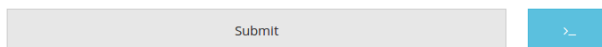
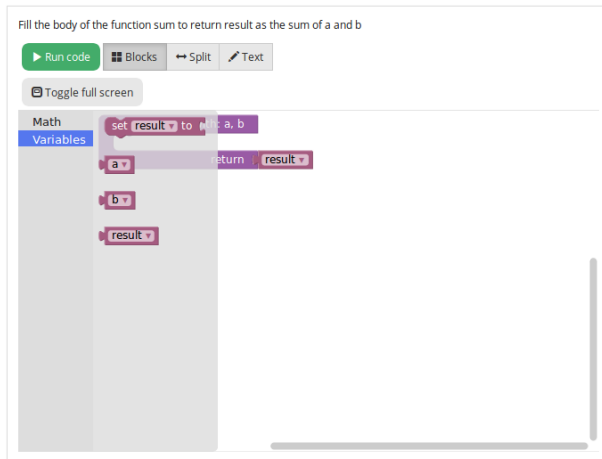
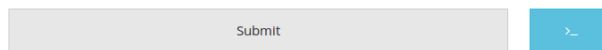
(suite de la page précédente)

```

<block type="variables_get">
  <field name="VAR">a</field>
</block>
<block type="variables_get">
  <field name="VAR">b</field>
</block>
<block type="variables_get">
  <field name="VAR">result</field>
</block>
</category>
</xml>

```

Here is the result from the student's point of view :

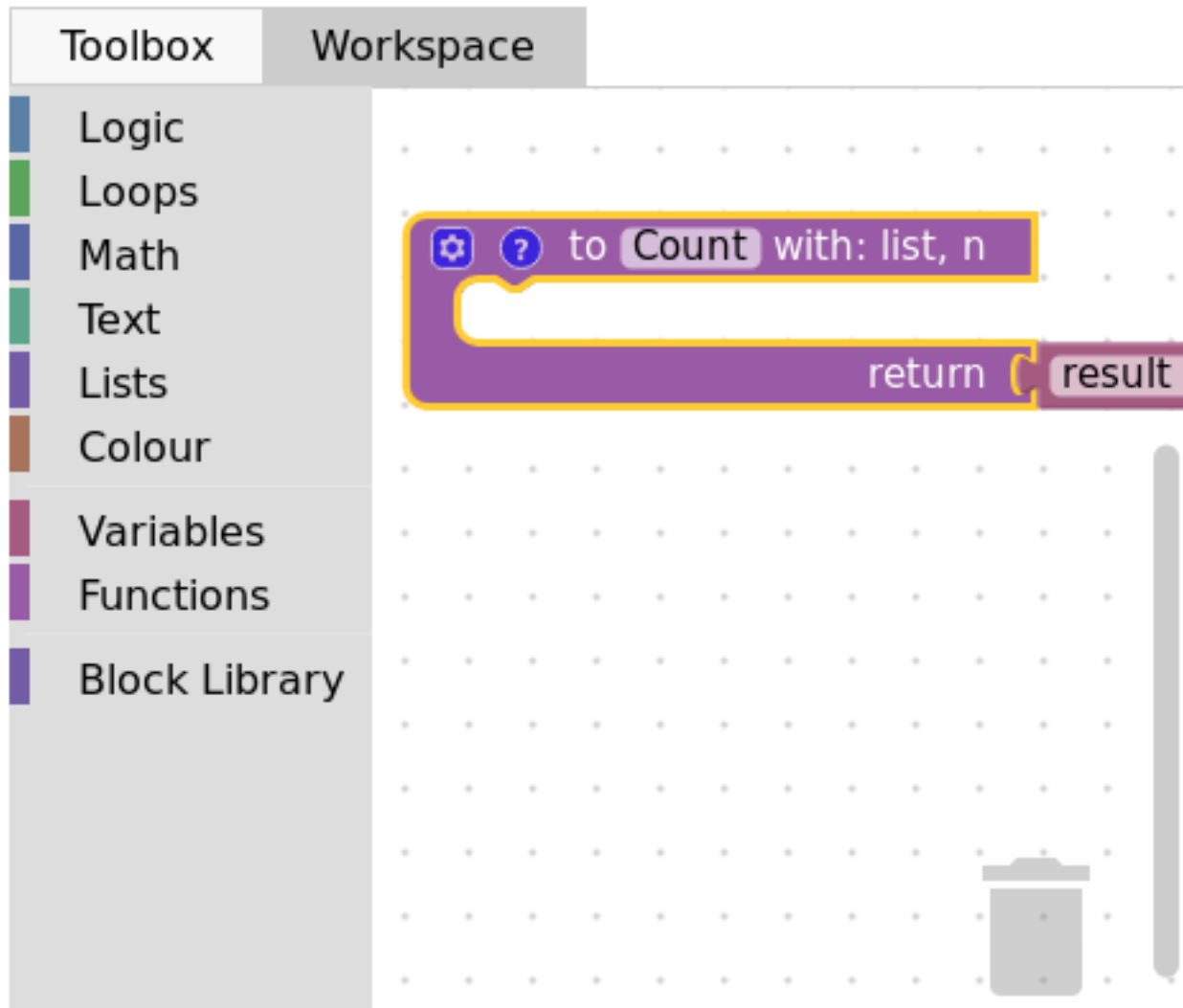


To get the full documentation about what can be achieved when modifying the toolbox manually, head to [this link](#) (Google documentation).

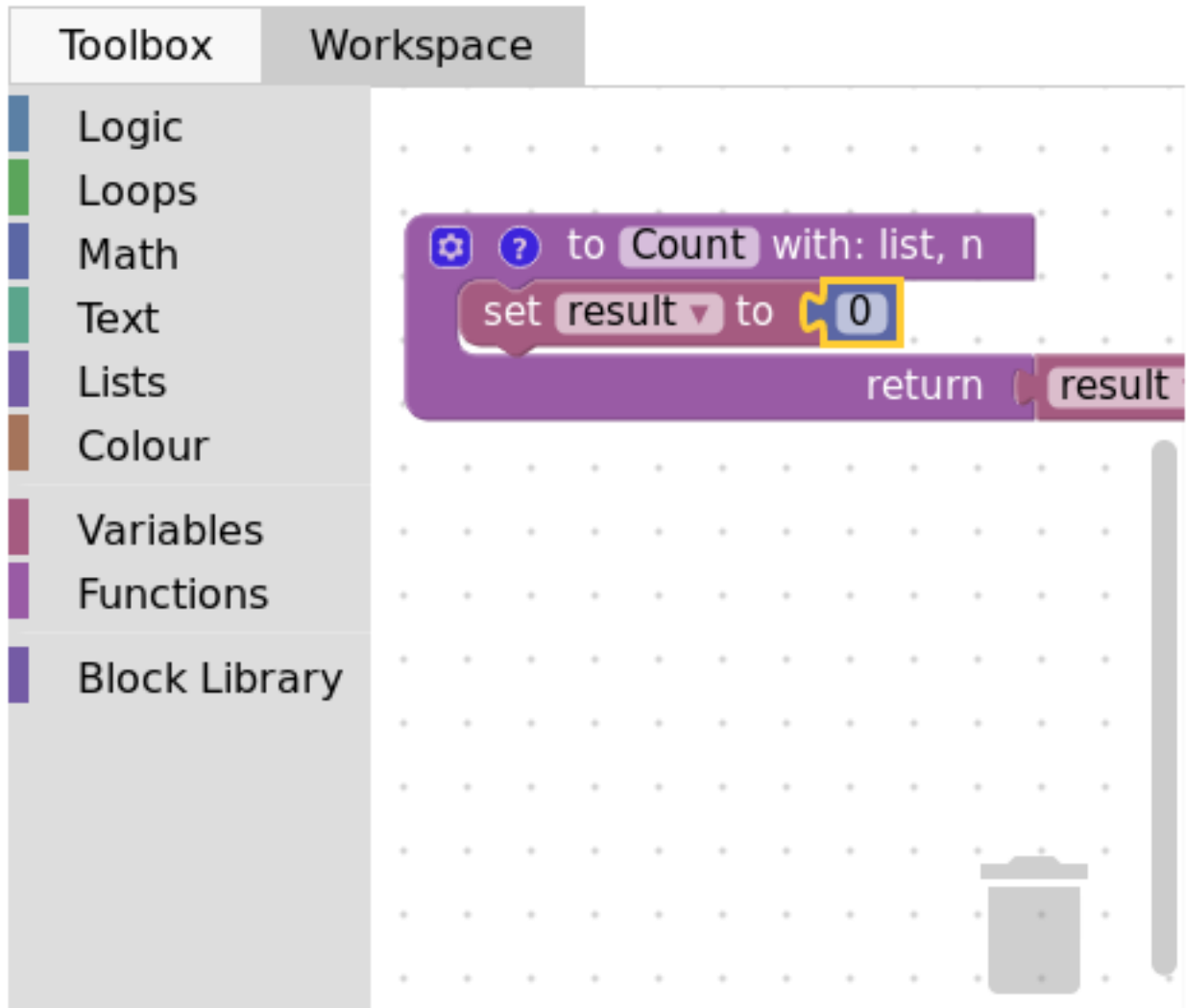
1.3 Example : an « only workspace » task

When creating a Blockly course, you might want your student to only re-order the blocks that are on the workspace rather than using a toolbox. This example will show you how to achieve that with the graphical interface. Here, we will take the very simple example of a function counting the number of occurrence of a number n in a list and returns it.

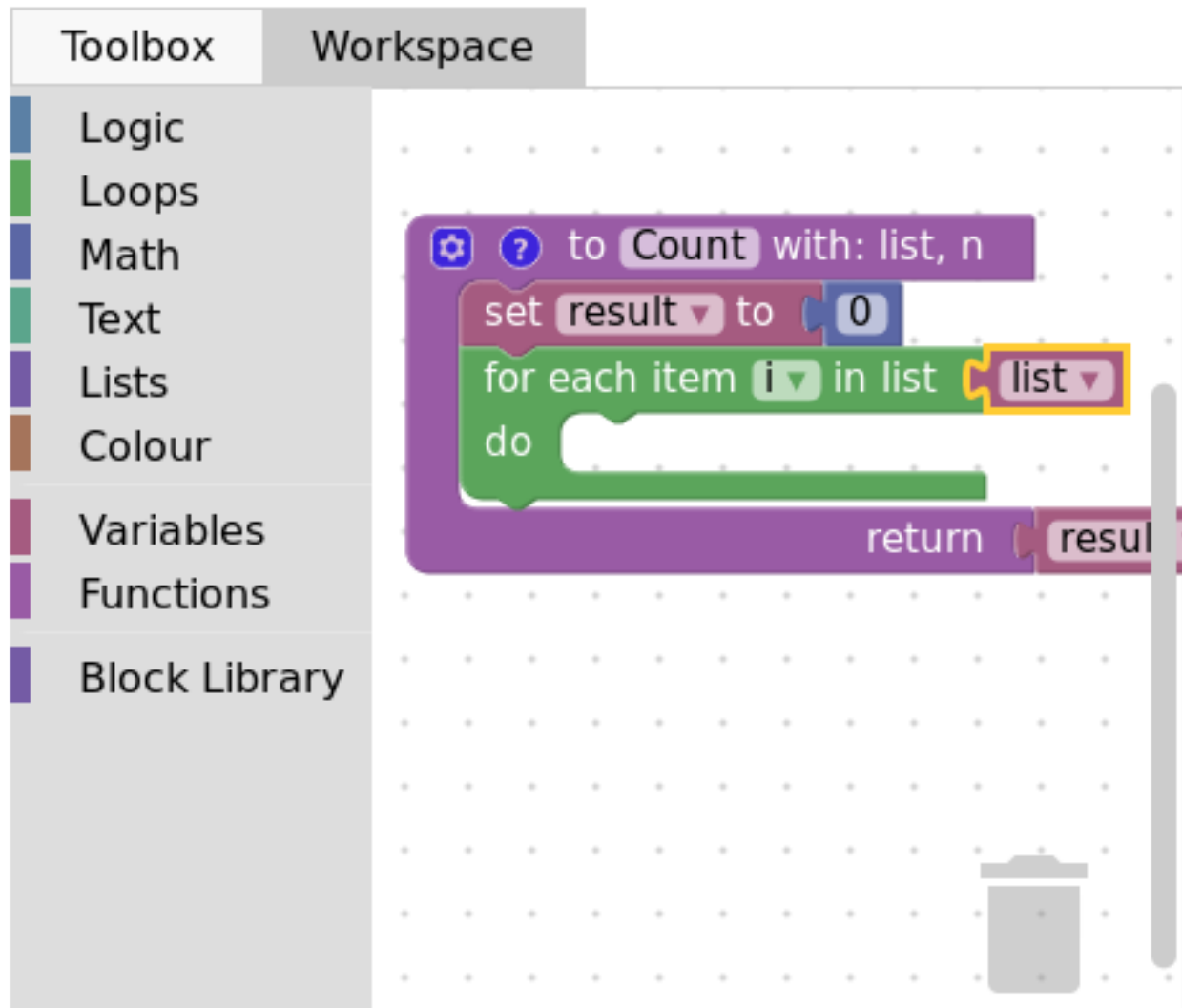
First, open the graphical editor, click on the workspace tab and create a function that takes two parameters *list* and *n*, and returns a value *return* (if you are not familiar with the graphical interface use, refer to [Example : create the sum function \(using the graphical interface\)](#))



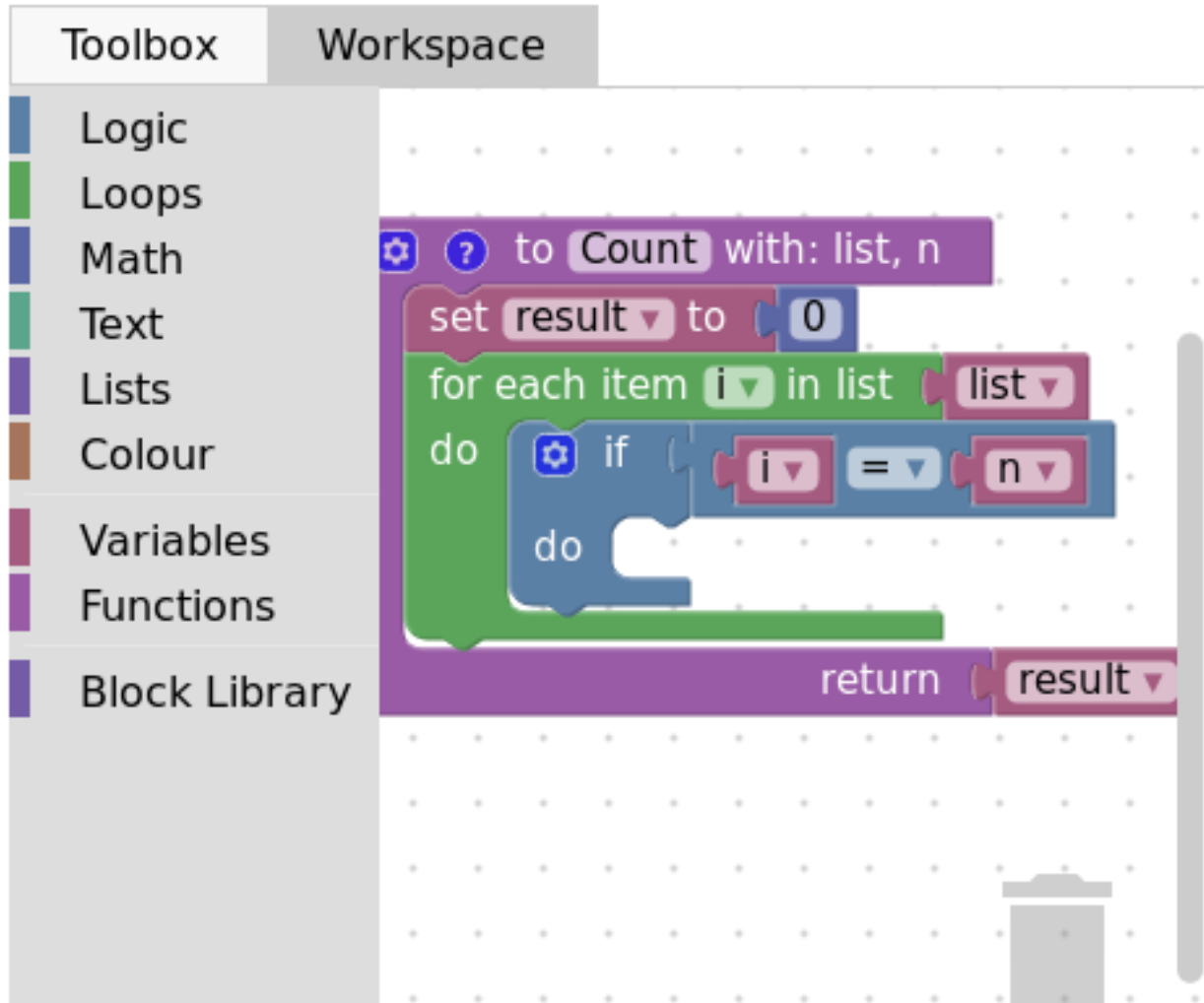
Then, from the *Variables* category, take the « set result to » block, and set it as the first block in the body of the function. From the *Math* category, get the « 0 » block, to first set result to zero. Here is the current progress :



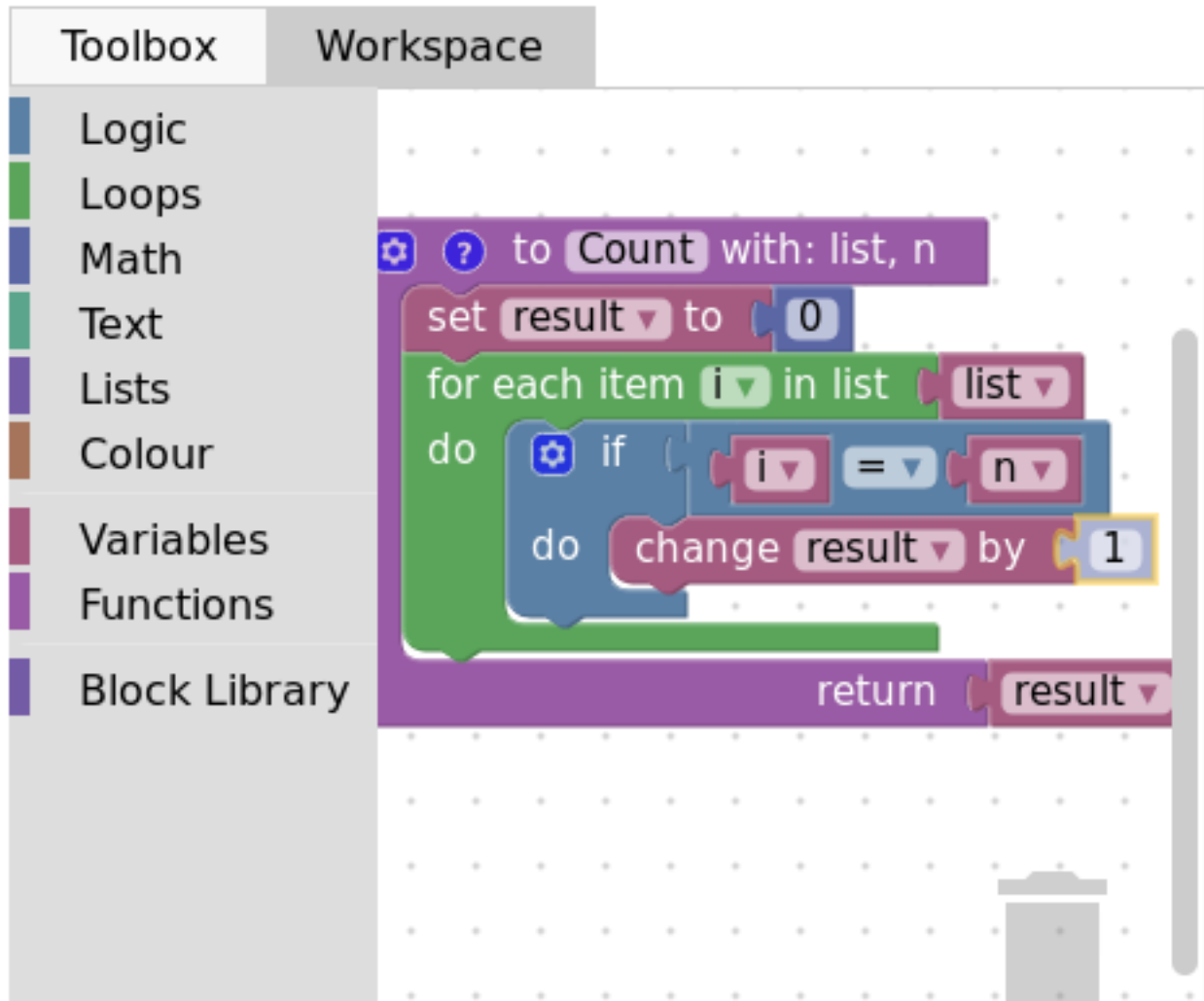
Next, from the *Loops* category, get the « for each item in list » block, plug it under the last one, and get the *list* variable to add it into the bloc :



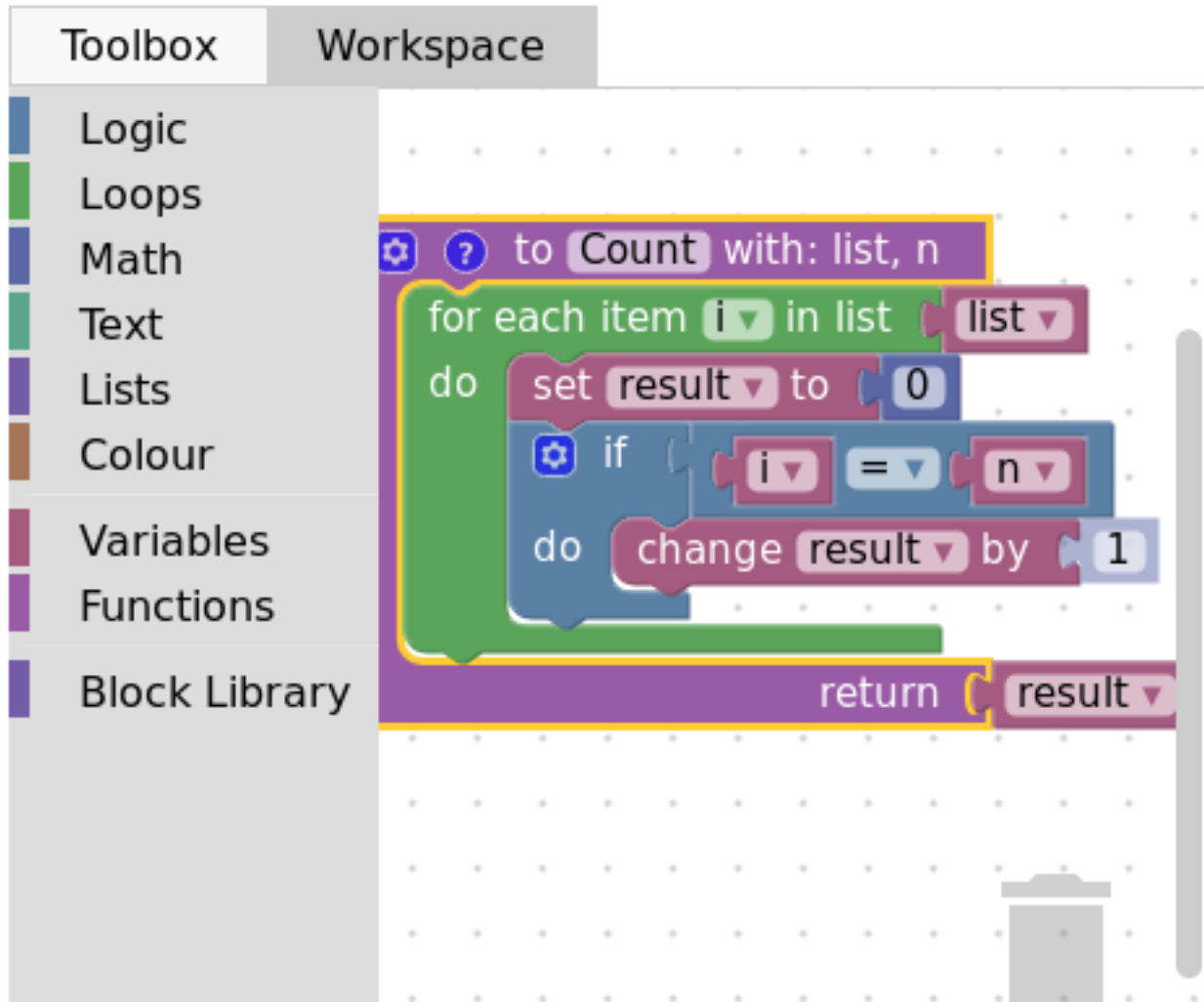
Add the « if » condition from the *Logic* category, and create our boolean $i == n$ with blocks from *Logic* and *Variables*



Finally, get the « change result by » block from the *Variables* sections and connect it to the body of the if. This is our correct function :



Now, we can purposefully add problems that the student will have to solve. We could change the boolean `==` to something else, or, in our case, move the « set result to 0 » block inside the loop body, like this :

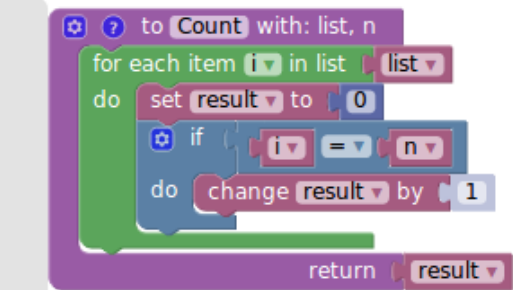


Here is what the student will see on INGINious :

Correct the code below

▶ Run code
 Blocks
 ↔ Split
 ✎ Text

🖥 Toggle full screen



Submit

>_

Again, we need to create a *run* file (same as the last one, will not be detailed here) and a correction file. Here is the code for the last one :

```
#!/bin/python3
# Open source licence goes here
from contextlib import redirect_stdout
import random

@@count@@

def countList(List, n):
    res = 0
    for i in List:
        if i == n:
            res += 1
    return res

if __name__ == "__main__":
    random.seed(55)
    for i in range(6): #6 tests
```

(suite sur la page suivante)

(suite de la page précédente)

```

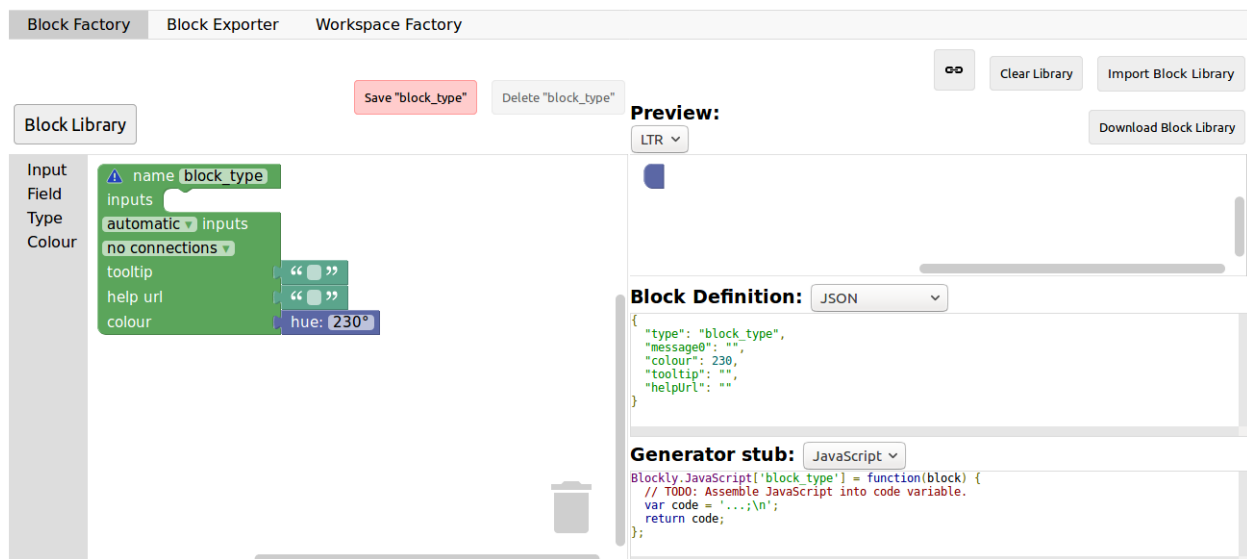
List = []
for j in range(15): #lists of 15 elements
    List.append(random.randint(0,10))
n = random.randint(0,10)
correct = countList(List, n)
output = Count(List, n)
if(correct != output):
    print("For the list "+str(List)+ " and the number "+str(n)+ " you have_
↪returned "
    + str(output) + " when the correct answer is " + str(correct) + ".")
    exit()
print("True")

```

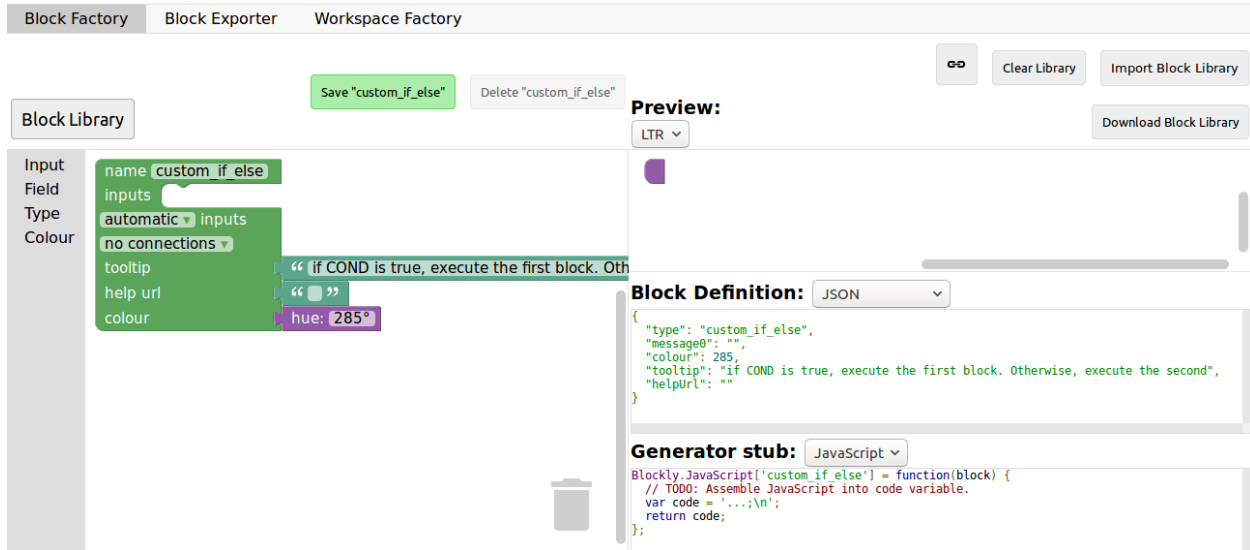
To make the correction and feedback easier, we defined a function giving the correct answer, and compare this function's result the the student one. We then run a few tests on random inputs. With the basic run file and this one in your task folder, it is complete.

1.4 Example : create a custom block (if/else)

If you feel like the existing blocks do not provide enough functionalities, you can create your own and export them. To do so, head to [this link](#), which is a factory allowing you to create new blocks using Blockly itself. This is the first screen :

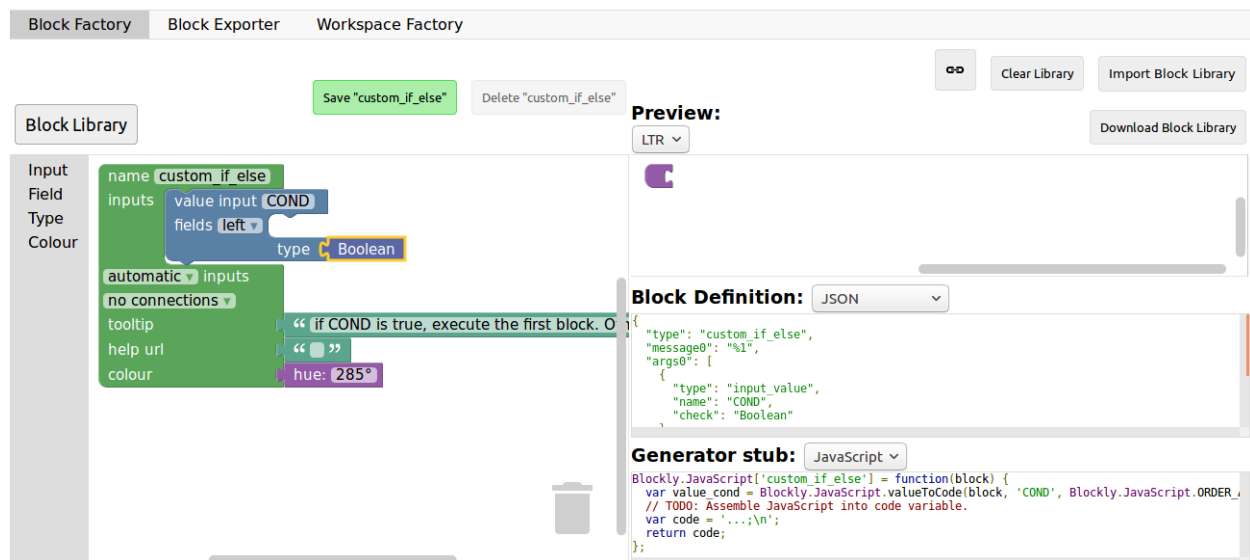


You will construct your block using the left side, while the right side is a live preview of both the visual and the code that will be generated. Let's construct an `if else` block. First, enter a name for it in the top field. It has to be unique across all Blockly blocks, so we will call it « `custom_if_else` ». Then, we can set a tooltip in the corresponding field, and pick a color for the block using the « `hue` » block (the color won't change the behavior).

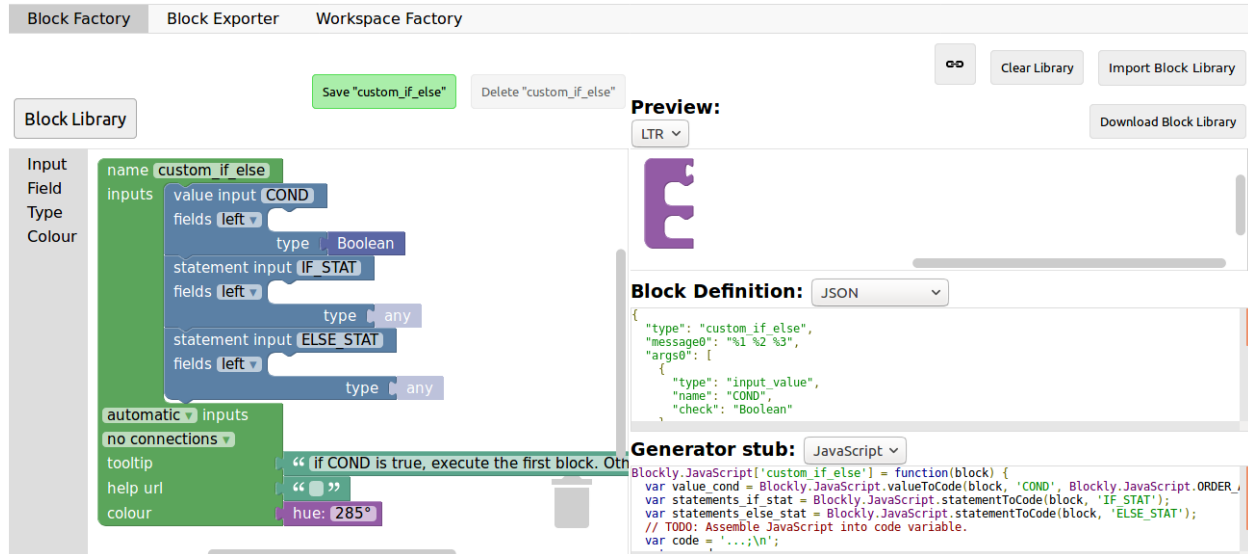


We will now construct the slots that our new block need. Since we are doing an `if else` we need to attach one boolean condition (the `if` condition), and two slots to put statements. This can be done with the *Input* category of the factory. There is three types of inputs : value, statement and dummy.

The value input create slots to the right of the block to plug in blocks that return a value, this is what we need for our condition. Each input needs to have a unique name across the block, and a type that is accepted. In our case, we name the input « `COND` » (capitals are a convention but not mandatory), and we set the type to *boolean* using the block in the category *Type*.

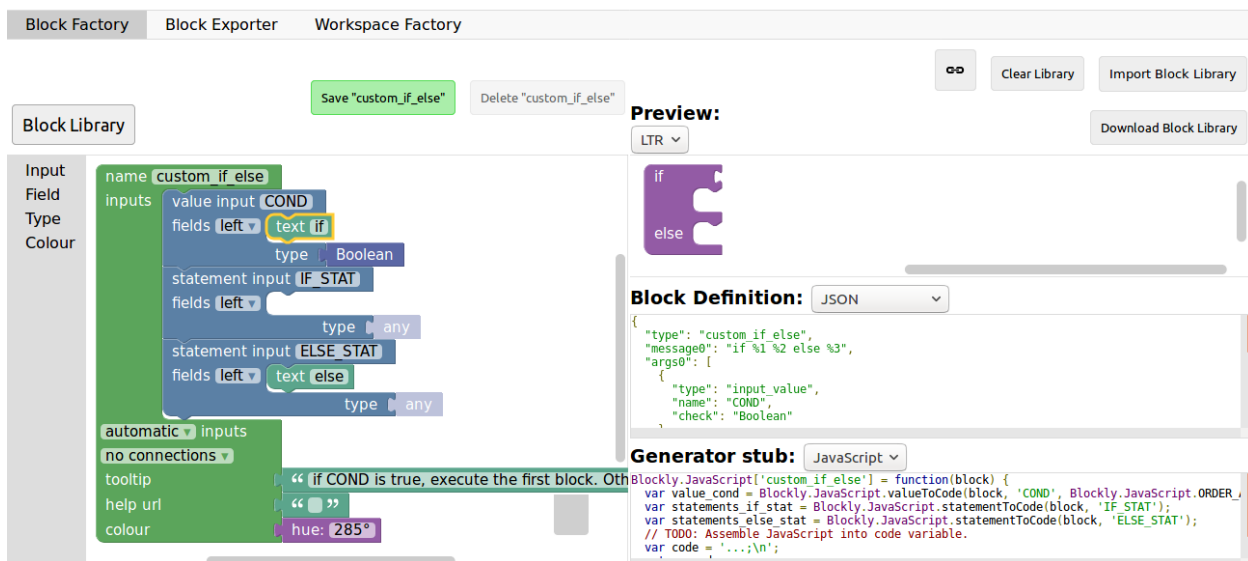


Now, we need the slots to put the statements. Again, click on the *Input* category and drag two *statements* blocks (dummy input won't be used in this tutorial, they simply allow to add extra space to a block for annotations but are not interactive). We need to name those inputs, respectively « `IF_STAT` » and « `ELSE_STAT` ».

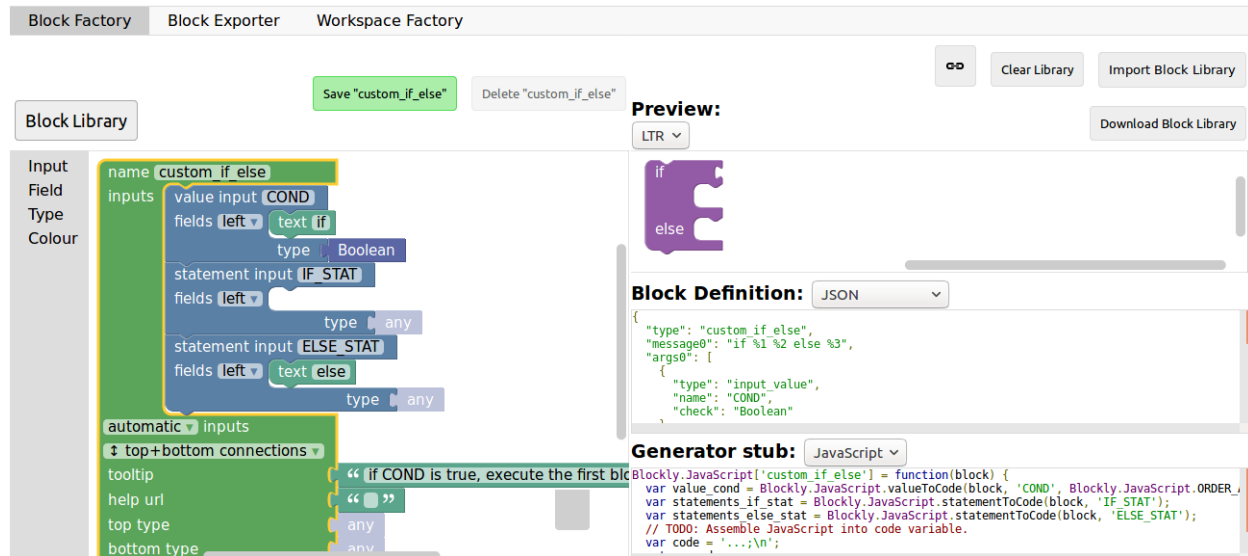


Now, our block has the correct structure, but adding text to it would make it clearer. This can be done using the *Field* category. There is a lot of different field items (user input, drop down, color pickers,...), to which you can find documentation [here](#).

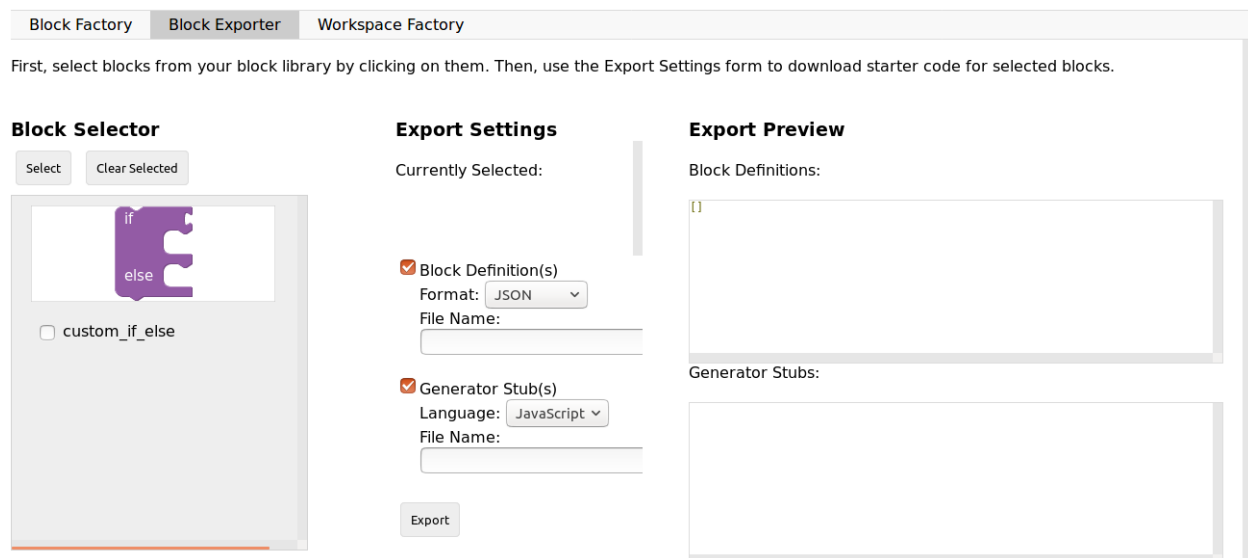
In our case, we need two *text* fields, one in the value input, and one in the second statement input. In the first field, we write « if », and in the second « else » (here, there is no need for the values to be unique).



Finally, we need to define the way our block interact with other using the connections drop-down list. Currently, *no connection* is selected, meaning that we can't plug the block into anything (this is the correct option for a function body for example). We need to be able to plug it into a block and to plug blocks after it, so we pick *top + bottom connections*, and here is our block done :



Now, we need to export it. First, click on the green Save "custom_if_else" button. Then, click on the Block Exporter tab :




Check the box next to our block name (this allows you to export multiple blocks at a time). We need the Python version of the code, so change the language of the generator and pick file names (here, *custom.json* and *custom.js*), then click Export :

Block Factory
Block Exporter
Workspace Factory

First, select blocks from your block library by clicking on them. Then, use the Export Settings form to download starter code for selected blocks.

Block Selector

Select Clear Selected



☒ custom_if_else

Export Settings

Currently Selected:

custom_if_else

☒ Block Definition(s)
Format: JSON
File Name: custom.json

☒ Generator Stub(s)
Language: Python
File Name: custom.js

Export

Export Preview

Block Definitions:

```
[{"type": "custom_if_else",
  "message0": "if %1 %2 else %3",
  "args0": [
    {
      "type": "input_value",
      "name": "COND",
      "check": "Boolean"
    },
    {
      "type": "input_statement",
      "name": "IF_STAT"
    },
    {
      "type": "input_statement",
      "name": "ELSE_STAT"
    }
  ]
}]
```

Generator Stubs:

```
Blockly.Python['custom_if_else'] = function(block) {
  var value_cond = Blockly.Python.valueToCode(block, 'COND', Blockly.Python.OF);
  var statements_if_stat = Blockly.Python.statementToCode(block, 'IF_STAT');
  var statements_else_stat = Blockly.Python.statementToCode(block, 'ELSE_STAT');
  // TODO: Assemble Python into code variable.
  var code = `...`;
  return code;
};
```

Save both files and you can close the tab, we will not use it anymore. To make it simpler, INGINious only uses one file to define all custom blocks, so we will need to copy over the code we downloaded. This is the general structure of the file we will create :

```
//License
'use strict';

Blockly.Blocks['block_name'] = {
  init: function() {
    this.jsonInit({
      //JSON code for the block
    });
  }
};

Blockly.Python['block_name'] = function(block) {
  //Generated code for the block
  //Custom code to represent the block
  return code;
};
```

Using the code we generated, we get :

```
//License
'use strict';

Blockly.Blocks['block_name'] = {
  init: function() {
    this.jsonInit({
      "type": "custom_if_else",
      "message0": "if %1 %2 else %3",
      "args0": [
        {
          "type": "input_value",
          "name": "COND",
          "check": "Boolean"
        },
        {
          "type": "input_statement",
          "name": "IF_STAT"
        },
        {
          "type": "input_statement",
          "name": "ELSE_STAT"
        }
      ]
    });
  }
};

Blockly.Python['block_name'] = function(block) {
  var value_cond = Blockly.Python.valueToCode(block, 'COND', Blockly.Python.OF);
  var statements_if_stat = Blockly.Python.statementToCode(block, 'IF_STAT');
  var statements_else_stat = Blockly.Python.statementToCode(block, 'ELSE_STAT');
  // TODO: Assemble Python into code variable.
  var code = `...`;
  return code;
};
```

(suite sur la page suivante)

```

        "type": "input_statement",
        "name": "IF_STAT"
    },
    {
        "type": "input_statement",
        "name": "ELSE_STAT"
    }
],
"previousStatement": null,
"nextStatement": null,
"colour": 285,
"tooltip": "if COND is true, execute the first block. Otherwise, execute the_
↪second",
"helpUrl": ""
});
}
};

Blockly.Python['block_name'] = function(block) {
    var value_cond = Blockly.Python.valueToCode(block, 'COND', Blockly.Python.ORDER_
↪ATOMIC);
    var statements_if_stat = Blockly.Python.statementToCode(block, 'IF_STAT');
    var statements_else_stat = Blockly.Python.statementToCode(block, 'ELSE_STAT');
    // TODO: Assemble Python into code variable.
    var code = '...\n';
    return code;
};

```

Now, we only need to link all the parts of our block into the corresponding python code. More details on how to get the code out of a block can be found on [this link](#). Here, we simply need to write the if/else structure around the part we already got in the variables and put it in a string :

```

Blockly.Python['block_name'] = function(block) {
    var value_cond = Blockly.Python.valueToCode(block, 'COND', Blockly.Python.ORDER_
↪ATOMIC);
    var statements_if_stat = Blockly.Python.statementToCode(block, 'IF_STAT');
    var statements_else_stat = Blockly.Python.statementToCode(block, 'ELSE_STAT');
    var code = 'if '+value_cond+" :\n"+statements_if_stat+" \nelse:\n"+statements_else_
↪stat+"\n";
    return code;
};

```

Now, we will save all that into a file, *custom_block.js*, and head to INGINious. First, create a new task and a Blockly subproblem, then copy your file into a public directory in your task (task_name/public). Refresh (F5) the task edition page to see you file. Then, on the corresponding subproblem, add your file name as « Additional block file » by clicking the blue button and typing the name of the file.

1-based index ☒

Scrollbars ☒

Sounds ☒

Files (Interpreter, Visual execution)

+ Add new file

Additional blocks files

File name

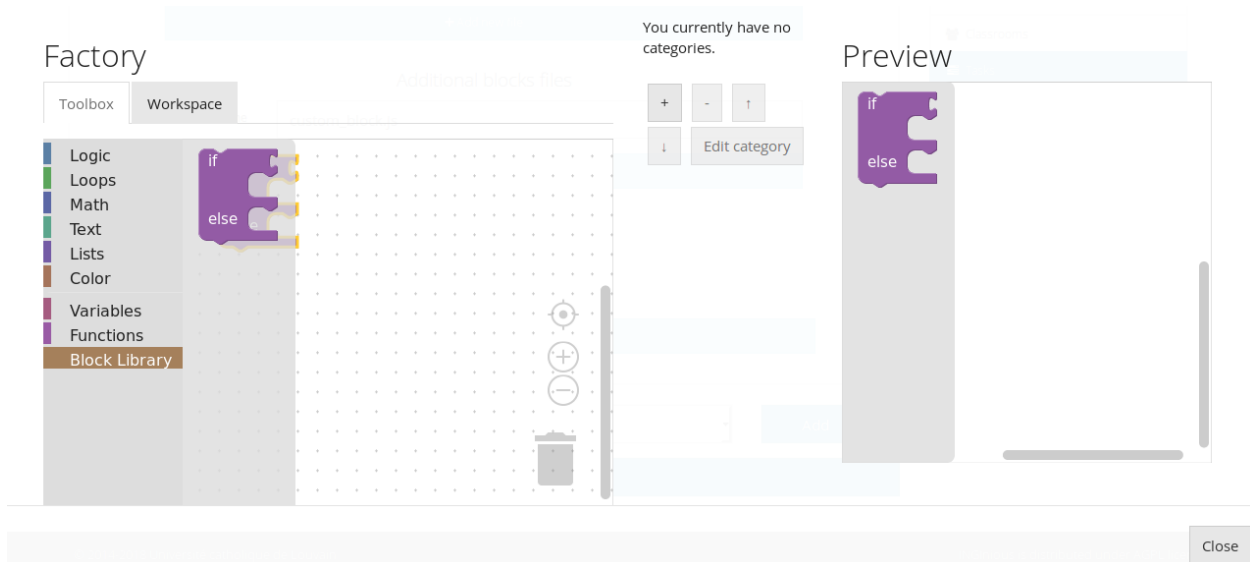
+ Add new file

Edit toolbox XML

Edit workspace XML

Edit toolbox/workspace graphically

Hit « Save changes » (top or bottom of the page), then refresh again. Now, you can use your block as any other to in your task, finding it under the *Block Library* category when using the graphical interface :



Comment créer une tâche labyrinthe ?

The easiest way to create such a task is to start from a very basic already existing task, and modify the files according to what we want. Zipped examples can be found at [this link](#) (currently, only the visuals with the pegman and the zombie are there, but more are to come). We will describe the content of each files further in this tutorial.

To start, unzip the files of your choice into the course directory. You can freely change the directory name (it must **not** contain spaces), the task name and instructions, etc. . .

Now, to edit the maze itself, open the file `maze.js` under `yourTask/public`. If you wish to change the graphical look of the maze, head to [this part](#) of the documentation to learn how to do so. If not, we will start by the shape of the maze itself.

Scroll to the part of the file that defines `Maze.map` : the table represents the current maze. Above, in the `Maze.SquareType` variable, you can see which number correspond to which type of maze tile. By default, we have :

- 0 is an empty tile (that might be populated randomly with non-interactive decor) where the character can't go
- 1 is an open tile where the character can walk
- 2 is the tile where the character starts the game
- 3 is the goal to reach to win the game
- 4 is a tile with an obstacle that will kill the character if it walks on it
- 5 is a special value used if you want to make the start and goal tile the same one

Now, let's look at our maze by default :

```
Maze.map =
[
  [0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 2, 1, 1, 3, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0, 0]
];
```

The maze is simply a straight line with three steps forward to take in order to win. We can see that more visually in the INGINious task :



Let's, for example, modify it to have a two empty tiles, then a turn to the goal and an obstacle where the previous goal was. Modify `maze.js` to this :

```
Maze.map =
[
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 2, 1, 1, 4, 0, 0],
  [0, 0, 0, 0, 3, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0, 0]
];
```

Hit save, reload the page and see the updated maze :



This is all the modifications that you will need to do in the `maze.js` file. The rest of the code handle the animations and visuals of the game, and should not be touched when creating new instances.

Now, to make our new maze correctable automatically, we need to edit the file `maze.tpl.py` (under the `student` directory) : the `MAP` variable should contain exactly the same array as the one we previously defined, which, in our case is :

```
MAP = [[0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 2, 1, 1, 4, 0, 0],
       [0, 0, 0, 0, 3, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0]]
```

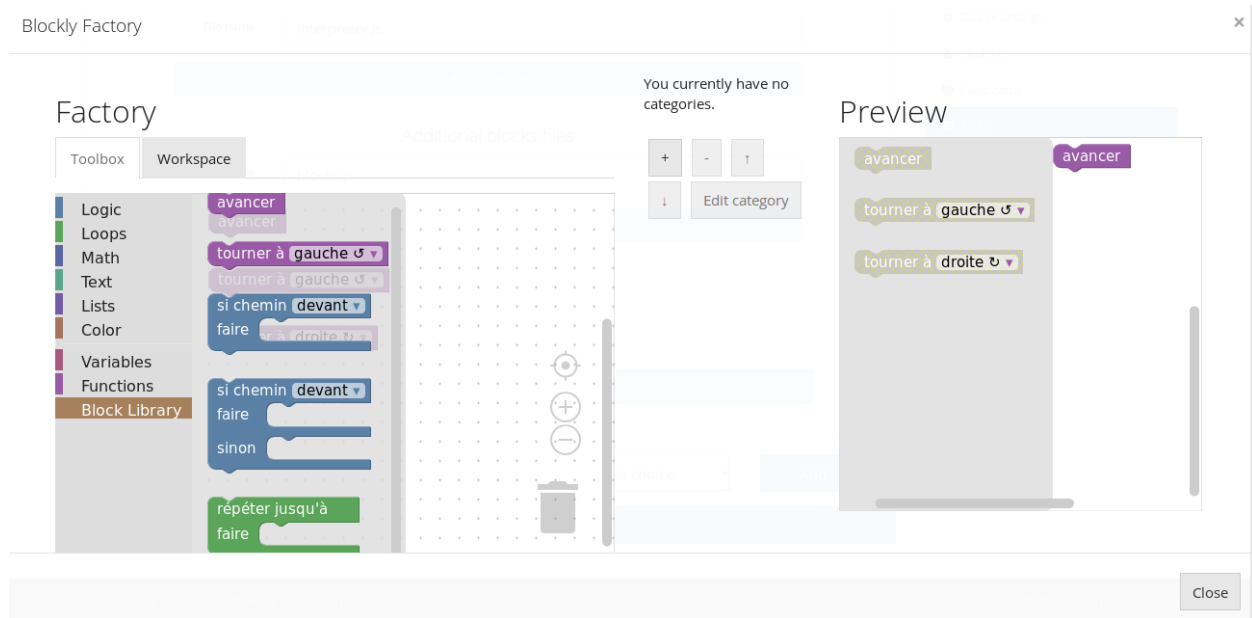
Then, change the line 106 to correspond to your subproblem name. By default, we have :

```
def student_code():
    @    @code@@
```

If our subproblem name is *example*, we need to change the value to :

```
def student_code():
    @    @example@@
```

The task should now work as expected. If you wish to add or remove blocks from the task, you can do so using the graphical user interface as you would for any other task. The blocks that are specific to a maze can be found under the *Block Library* category, and are defined in the file `blocks.js` (under `public`), which should not be modified, except if you want to add new custom blocks to it.



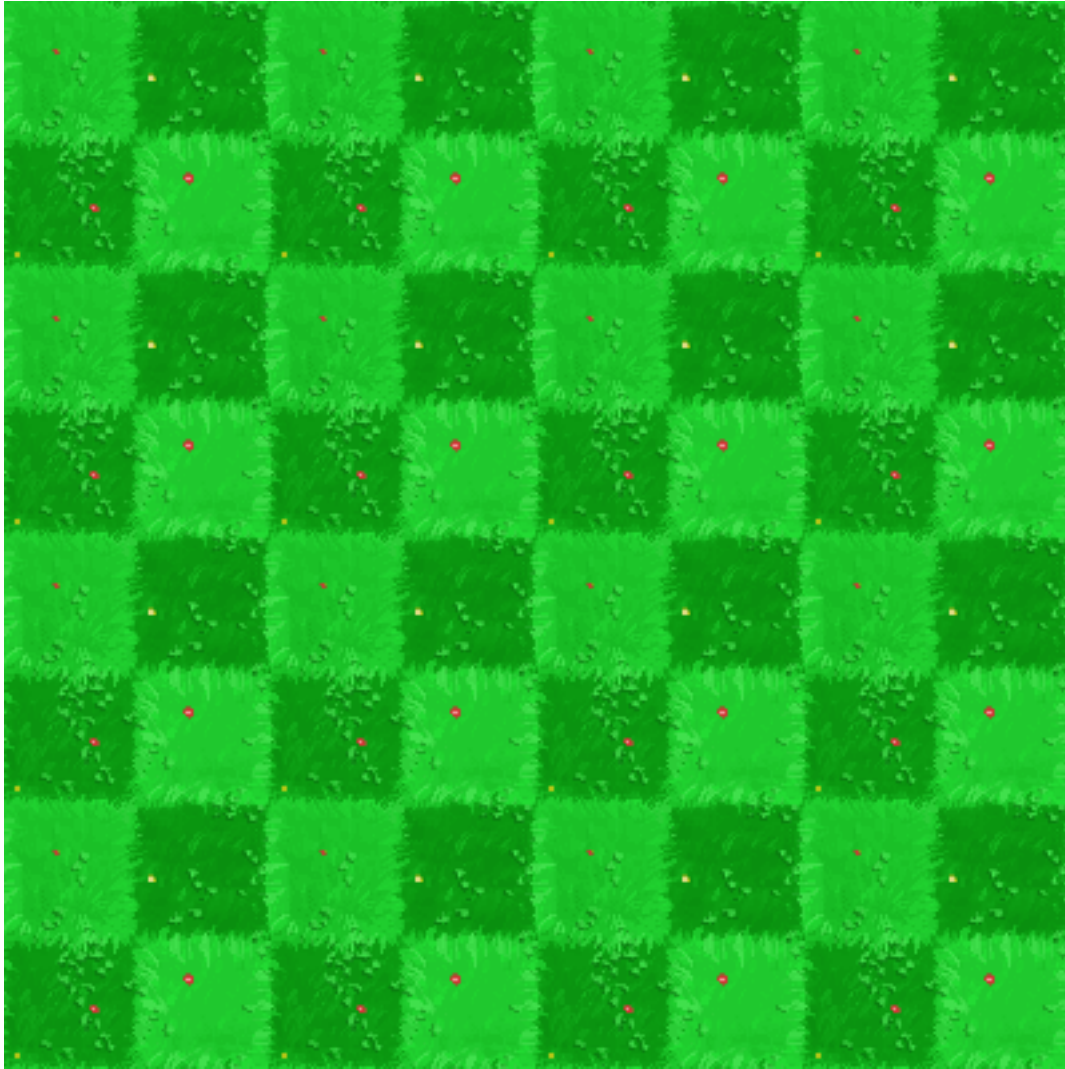
For the last files that were not yet mentioned, `interpreter.js` deals with internal animations, and should not be modified, as well as `run.py` (at the root of the task), which is simply a classical run file, as described earlier in this documentation.

Comment changer le visuel du labyrinthe

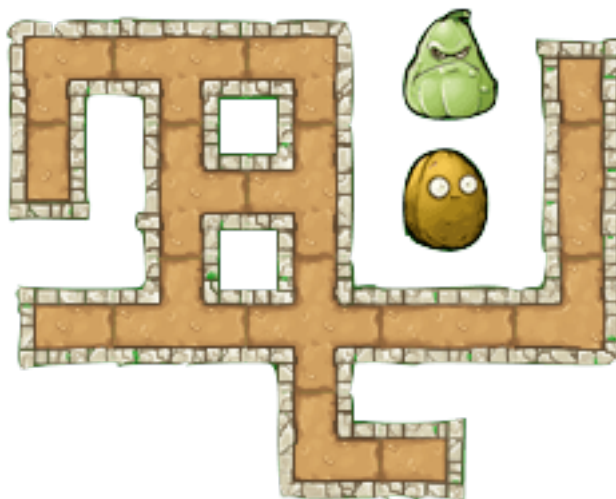
3.1 Files you will need

First, you will need all those files, in the exact specified format.

A background representing an 8x8 board, in png format, for example the following :



A file representing all possible intersections of the path and, if wanted, two « passive » entities to populate the map in png.



A sprite for the avatar in png, with the following format : 21 frames of equal size and equally spaced, with the character in the center of each one. The frames need to be ordered this way :

- The first (left) sprite is the character from the back. Then, we have a transition where it turns on it's right in three steps
- The next frame, the 5th one, is the character on it's right profile. Then, another transition with a turn to the right in three steps
- The 9th frame is then the character from the front. Again, the three next are a transition to the right.
- The 13th frame is the left profile, followed by the last transition, getting us to 16 images
- Finally, 5 images representing a walking animation (currently not used by the application, you can put any frame) to get to the 21.



A marker representing the goal of the maze idle, when the character is on another tile. As every other « interactive » part of the maze, it is usually in gif, but a png would work as well

The marker when it is reached by the character and the game is won (gif or png)

An obstacle idle, when the character is on another tile (gif or png)

The obstacle when killing the character, ending the game (gif or png)

You may also add, but do not need to :

- mp3 and ogg files with a sound to be played when the character hit an obstacle
- mp3 and ogg files with a sound to be played when the character loses the game without hitting an obstacle
- mp3 and ogg files with a sound to be played when the character wins the game

All of those files need to be put in the task folder, under : `taskName/public/maze`

3.2 Files to modify

The only file to modify is `maze.js`, that you will find under `taskname/public`. At the beginning (line 37 to 52), you will have to tweak a few parameters of the variable `Maze.SKIN` and to replace the names of the files with your own files, like so :

```
Maze.SKIN = {
  sprite: task_directory_path + 'maze/avatar.png',
  #Rest of the parameters
}
```

Will become :

```
Maze.SKIN = {
  sprite: task_directory_path + 'maze/myAvatarName.png',
  #Rest of the parameters
}
```

Here is a break down of all the variables and what they correspond to :

- `sprite` : your avatar
- `tiles` : the tiles to show the paths
- `marker` : the marker or goal of the maze when idle

- goalAnimation : the marker when the game is won
- obstacleIdle : the obstacle when idle
- obstacleAnimation : the obstacle killing the character
- obstacleScale : the scale of the obstacle regarding the maze, the higher the bigger
- background : the background for the paze (png)
- graph and look, no need to modify
- obstacleSound : both sounds file for when the character hits an obstacle or [" "] if you don't want any
- winSound : both sounds file for when the character wins de game or [" "] if you don't want any
- crashSound : both sounds file for when the character looses the game or [" "] if you don't want any
- crashType : to not modify

CHAPITRE 4

Indices et tables

- genindex
- modindex
- search