

---

# **BlocklyTasks Documentation**

**Celine Deknop**

**Jul 31, 2018**



---

## Contents

---

<b>1</b>	<b>Creation of a <i>basic</i> Blockly task</b>	<b>1</b>
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
<b>2</b>	<b>Blockly visual tasks</b>	<b>29</b>
2.1	Available types of tasks . . . . .	29
2.2	Create a new maze task . . . . .	30
2.3	Create a new collect/create task . . . . .	34
2.4	Create a new drawing task . . . . .	37
<b>3</b>	<b>How to change the maze’s visuals</b>	<b>43</b>
3.1	Files you will need . . . . .	43
3.2	File to modify . . . . .	46
<b>4</b>	<b>Indices and tables</b>	<b>49</b>



# CHAPTER 1

---

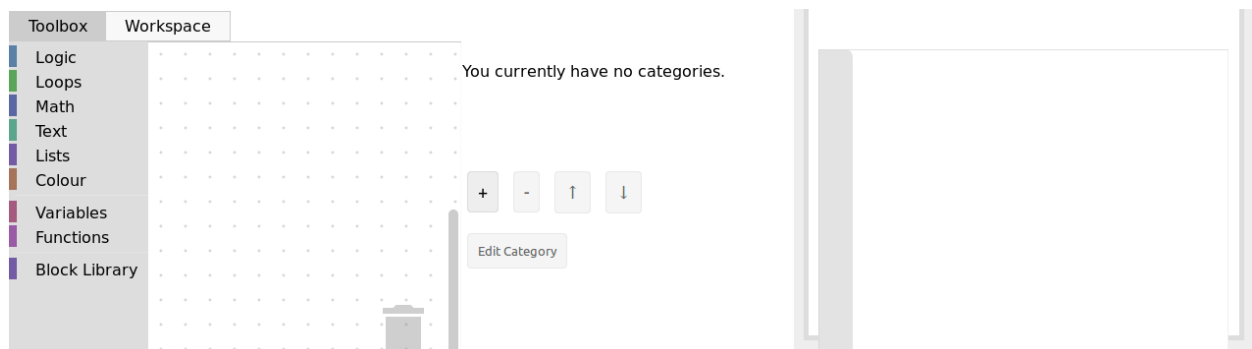
## Creation of a *basic* Blockly task

---

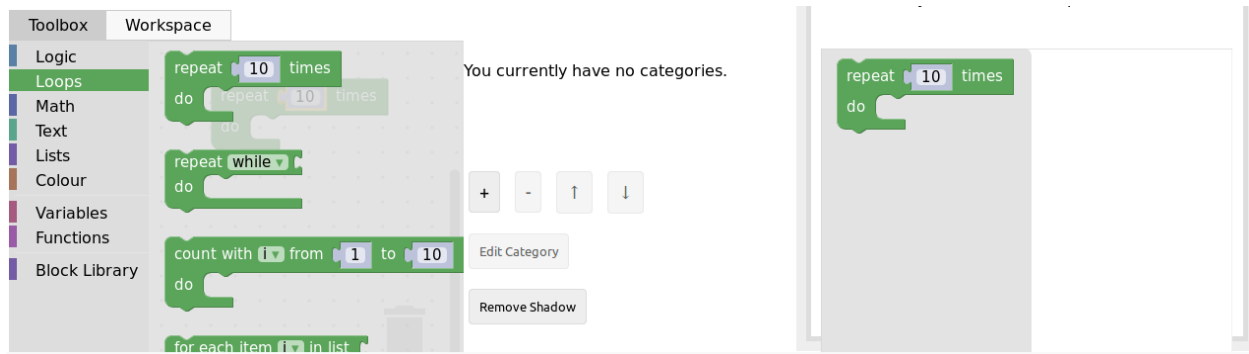
There is, first, a few steps that are the same as for any other tasks. Those are :

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”)

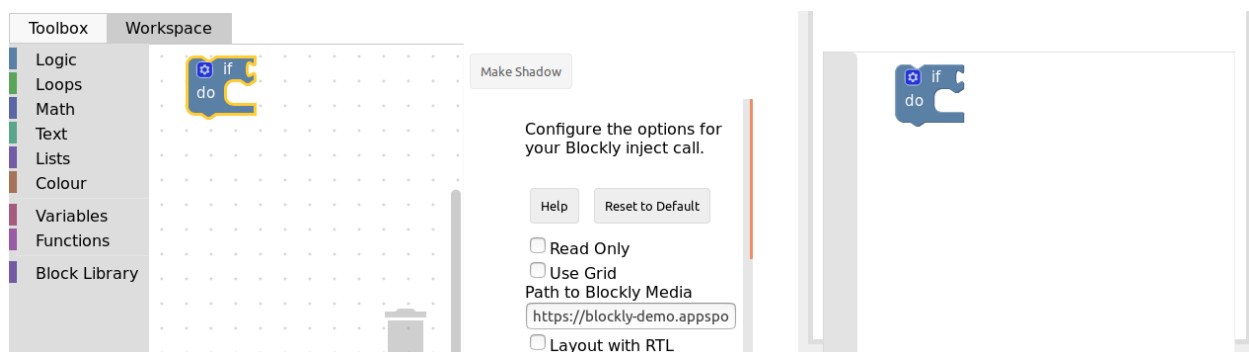
Now, there is two ways to configure Blockly : either using the embedded graphical interface or by entering the blocks by hand. Since the first solution is more beginner-friendly, let’s explore it first. Scroll down all the way and click “edit toolbox/workspace graphically”. This is what you will see.



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.

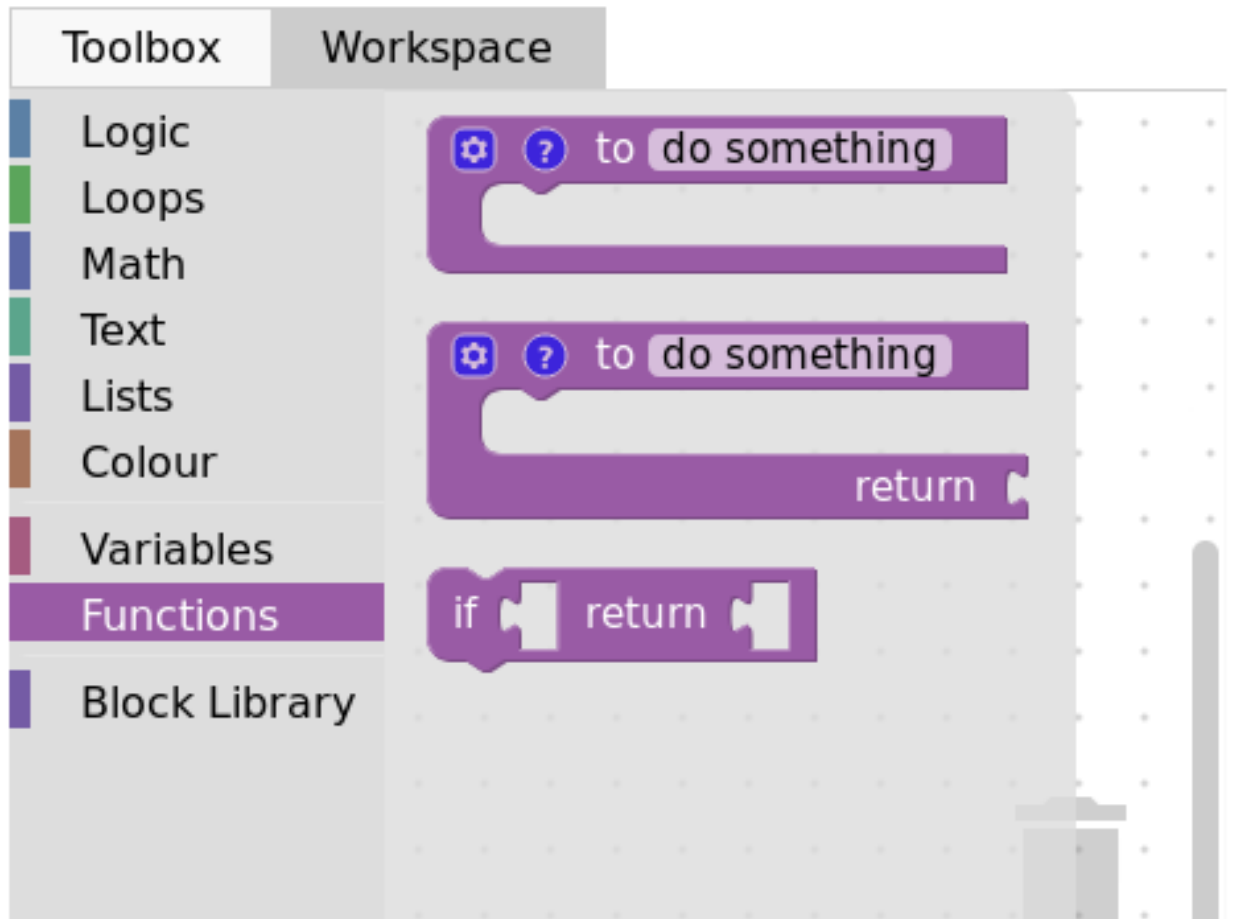


Let’s now see an example of what can be done for a simple exercise.

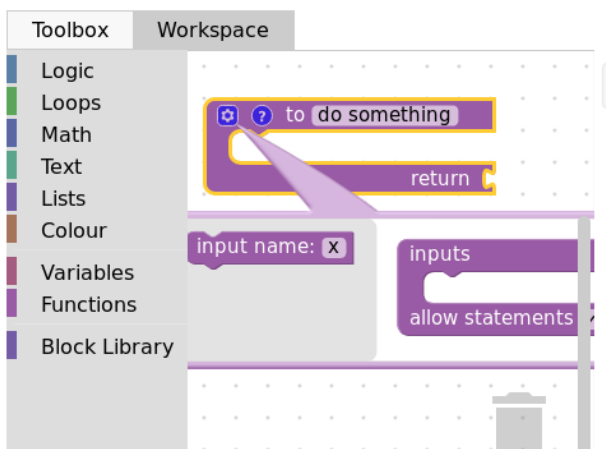
### 1.1 Example : create the sum function (using the graphical interface)

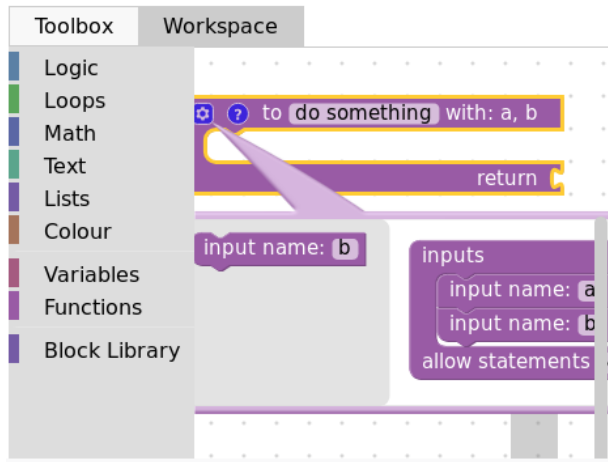
Here, we are in the case where we want the student to create a function, which means we have to provide him with it’s signature in the workspace. Our Sum function needs to take in two parameters, the two numbers to sum (let’s call them a and b), and return the resulting sum.

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.

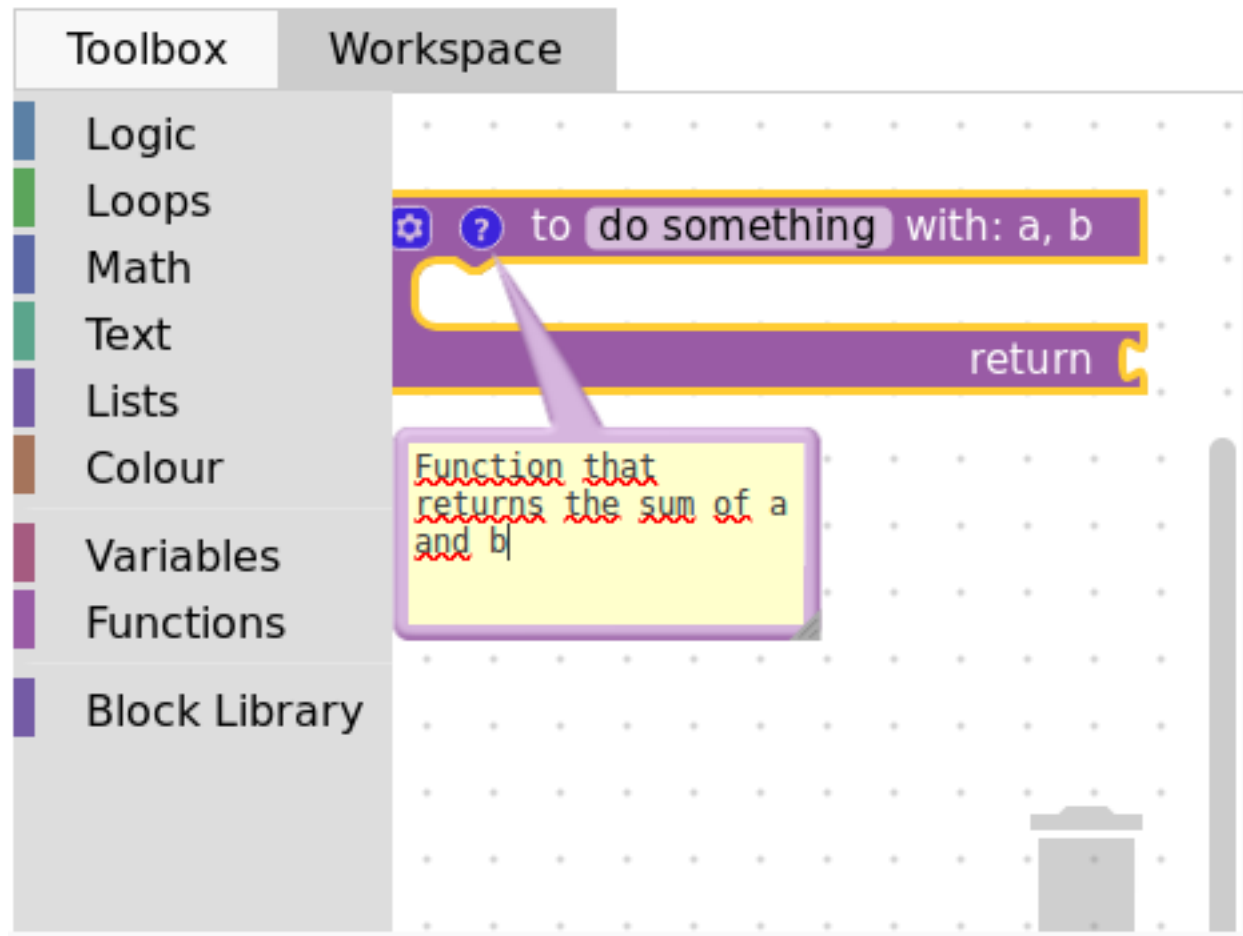


Now, configure the function. The wheel icons allow us to add parameters. Simply name your parameter ( $x$  by default), then connect the block into the right space, like so :



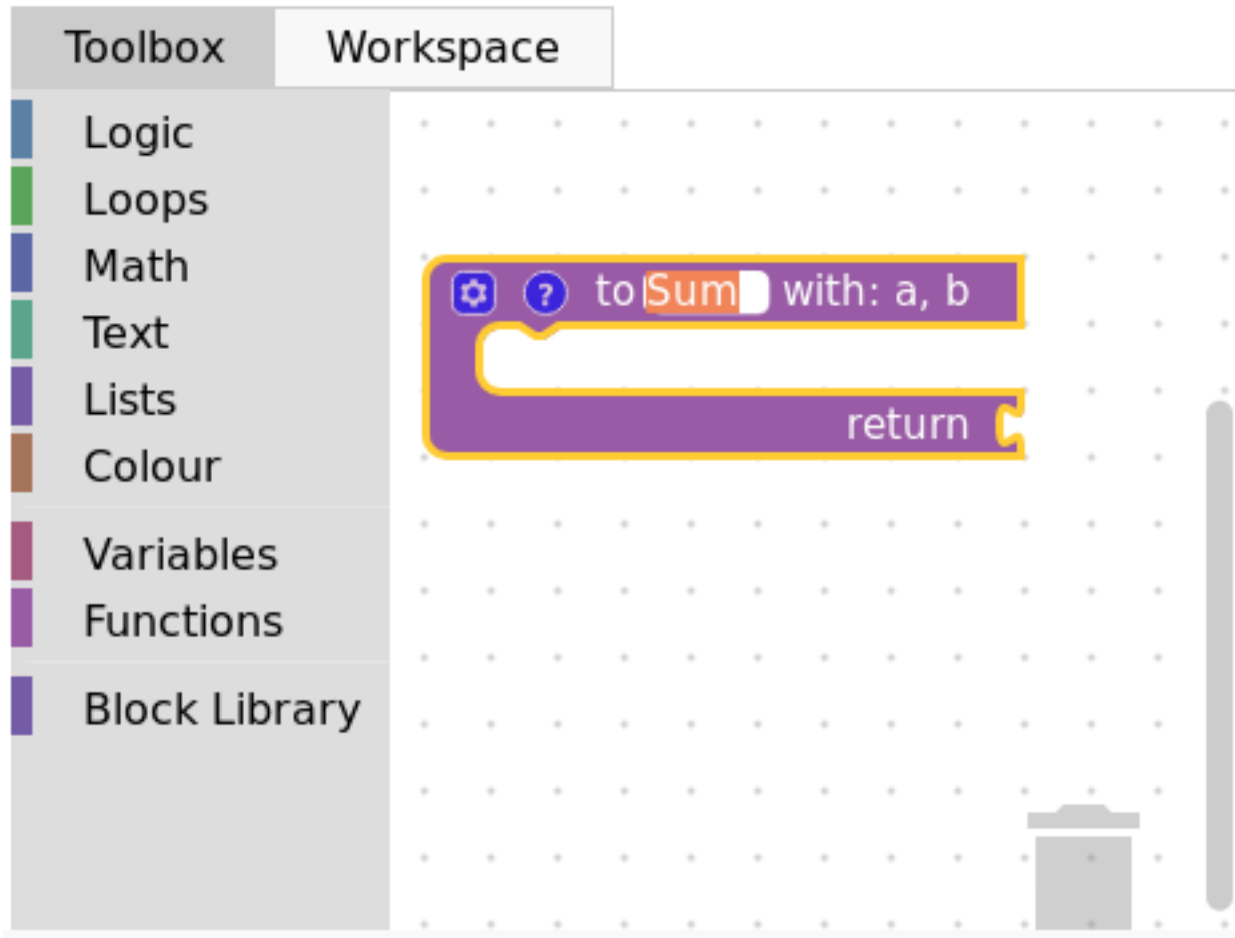


The ? icon allows us to set a tooltip (text that show on mouseover) simply by typing in the field :

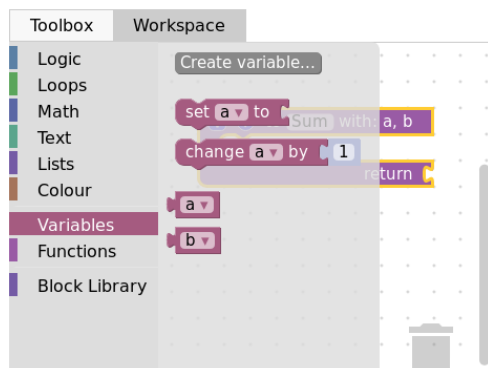


Finally, we have to name our function, changing the *do something* into what we want, here, *Sum* :

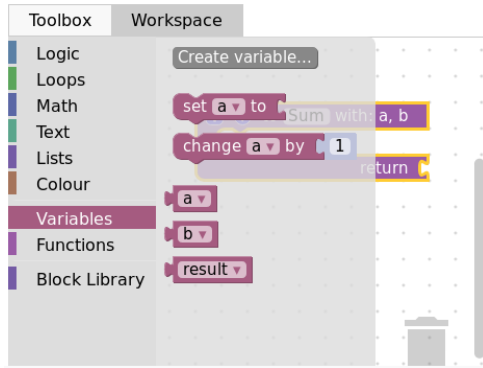




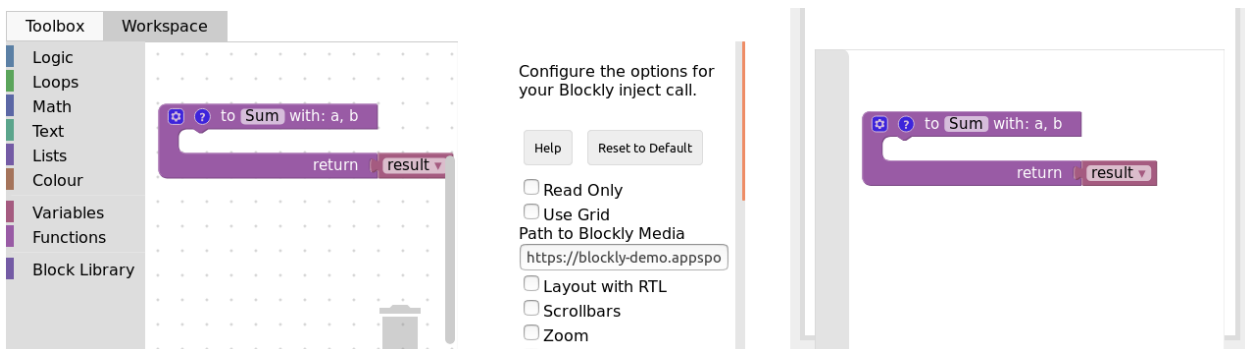
Now, let's create a variable to hold the result. Click on the "Variables" category and select "create variable". Input your variable name, "result" for example, and it will be available in the category :



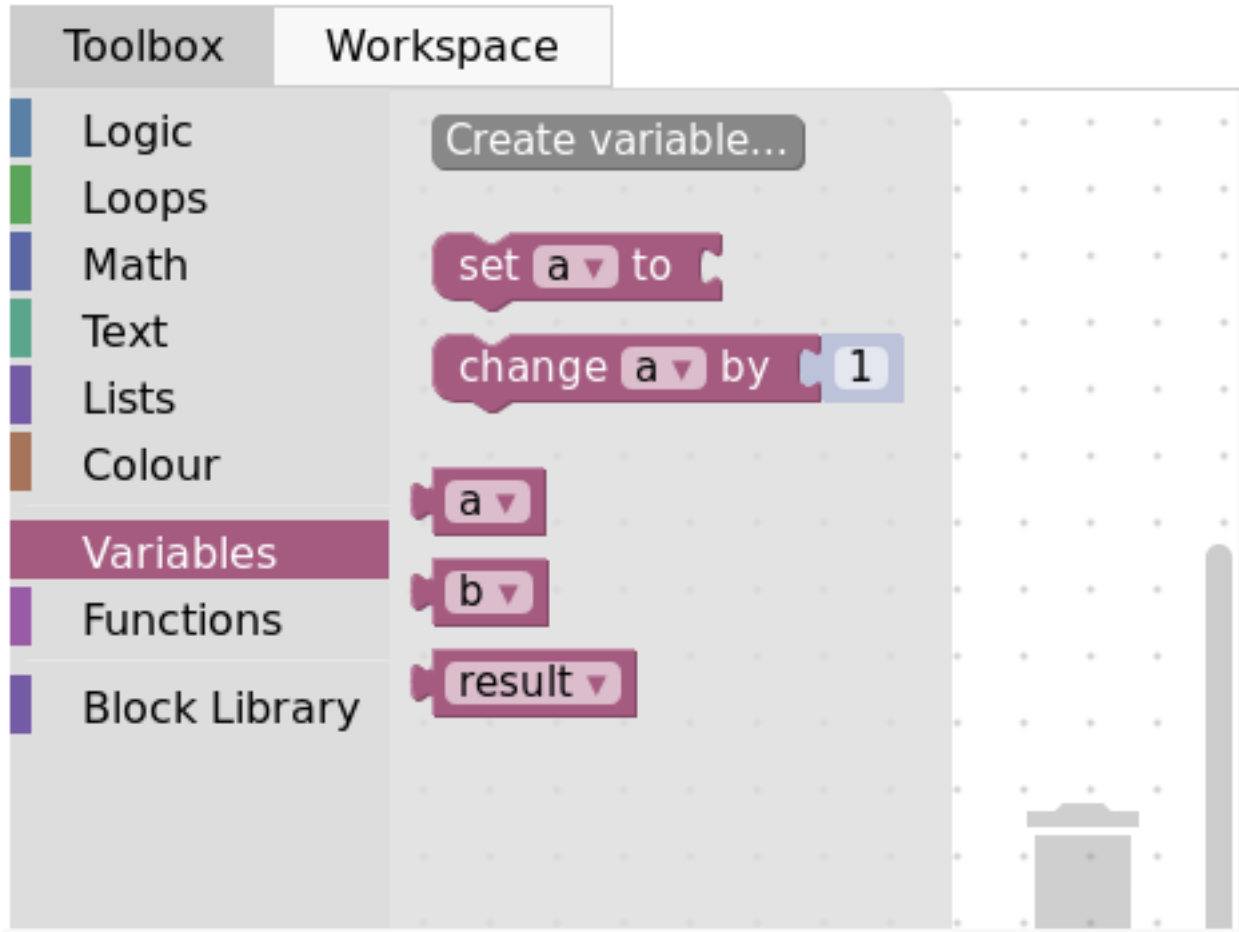
New variable name:



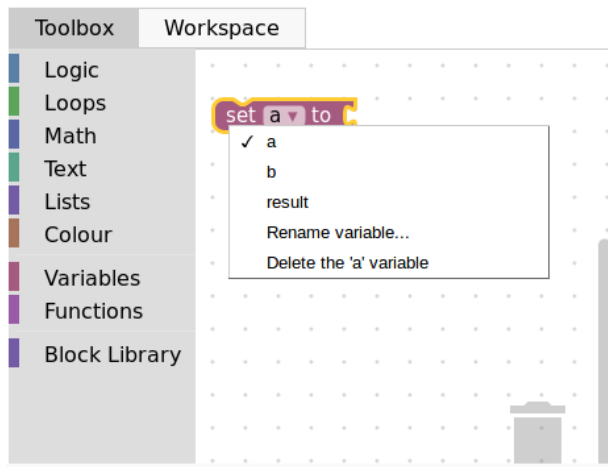
Finally, select the corresponding block and plug it into the “return” spot. Here is our basic workspace done, with the preview :

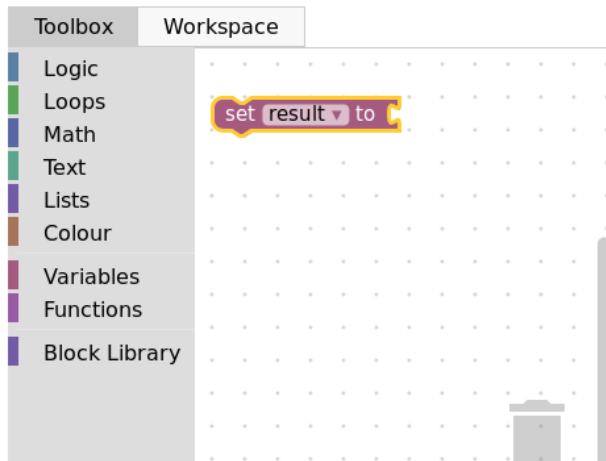


Now, it is time to create the toolbox. Click on the corresponding tab, and select the blocks that you want for the task. In our case, we first need to re-create all the previous variables, the same way as we did for the *result* one (clicking on create variable). Here is what we end up with :

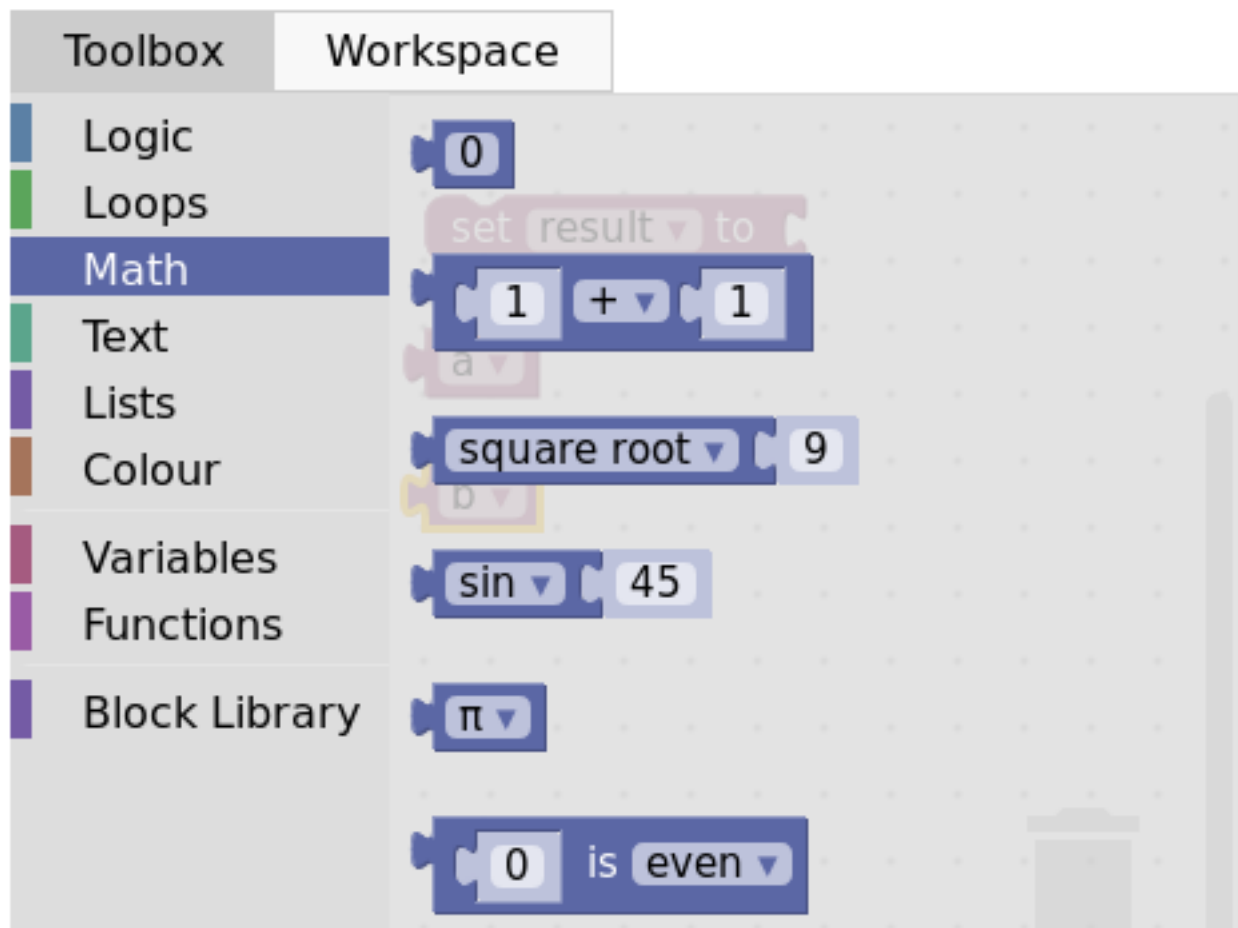


Then, we want the *set* block, so we drag it to the toolbox. Using the arrow next to the variable name, we can select the variable we want by default (*result* in our case) :

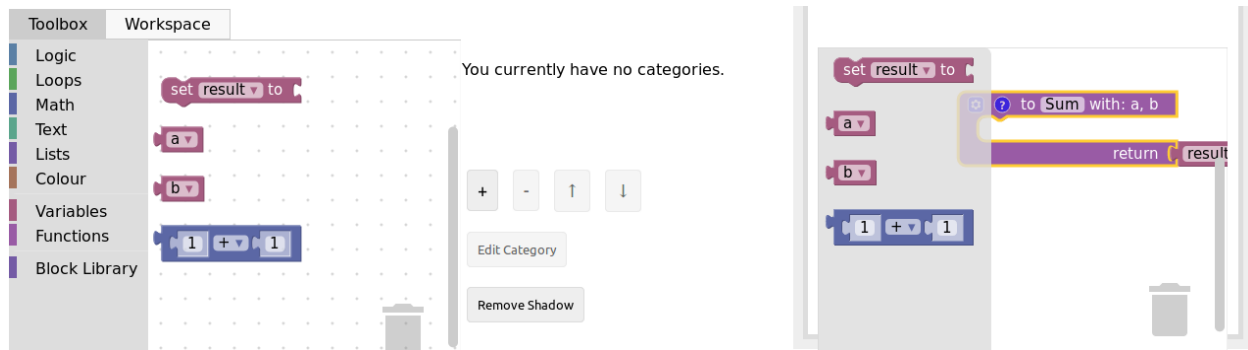




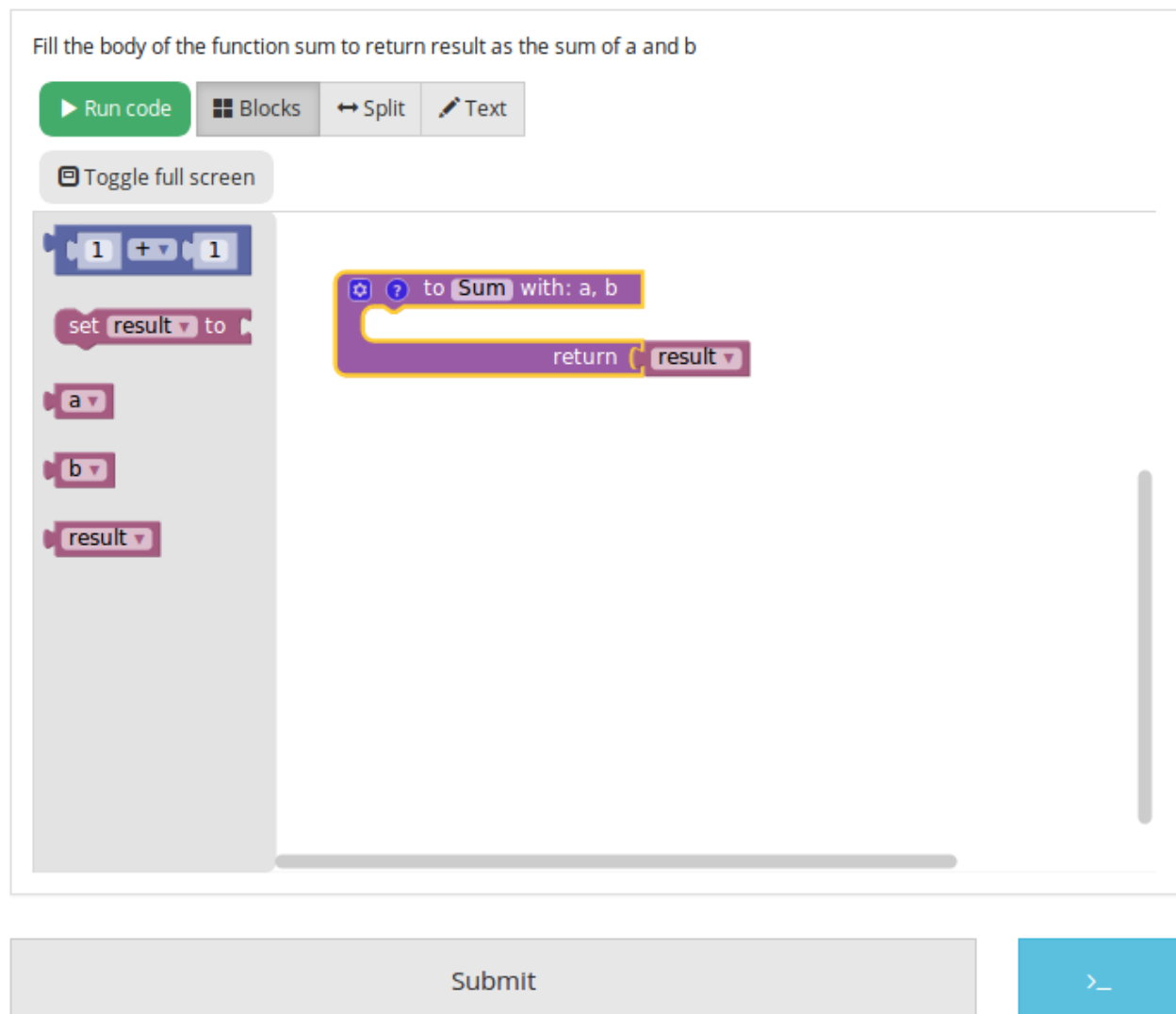
Then, we add the two previously created variables “a” and “b” as well. Finally, we want the sum operator from the math category :



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>
```

(continues on next page)

(continued from previous page)

```

<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>

```

(continues on next page)



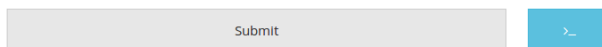
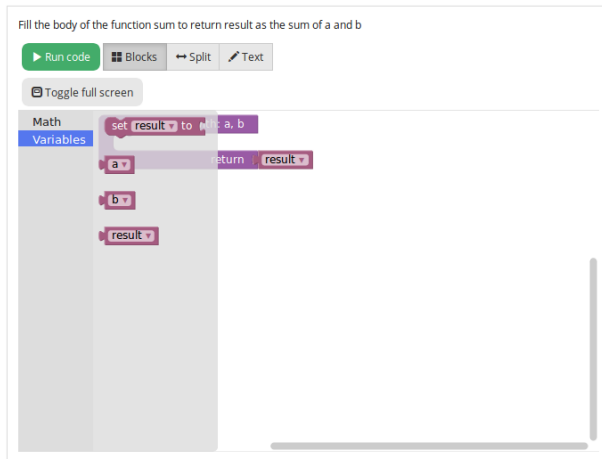
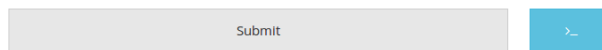
(continued from previous page)

```

<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 :

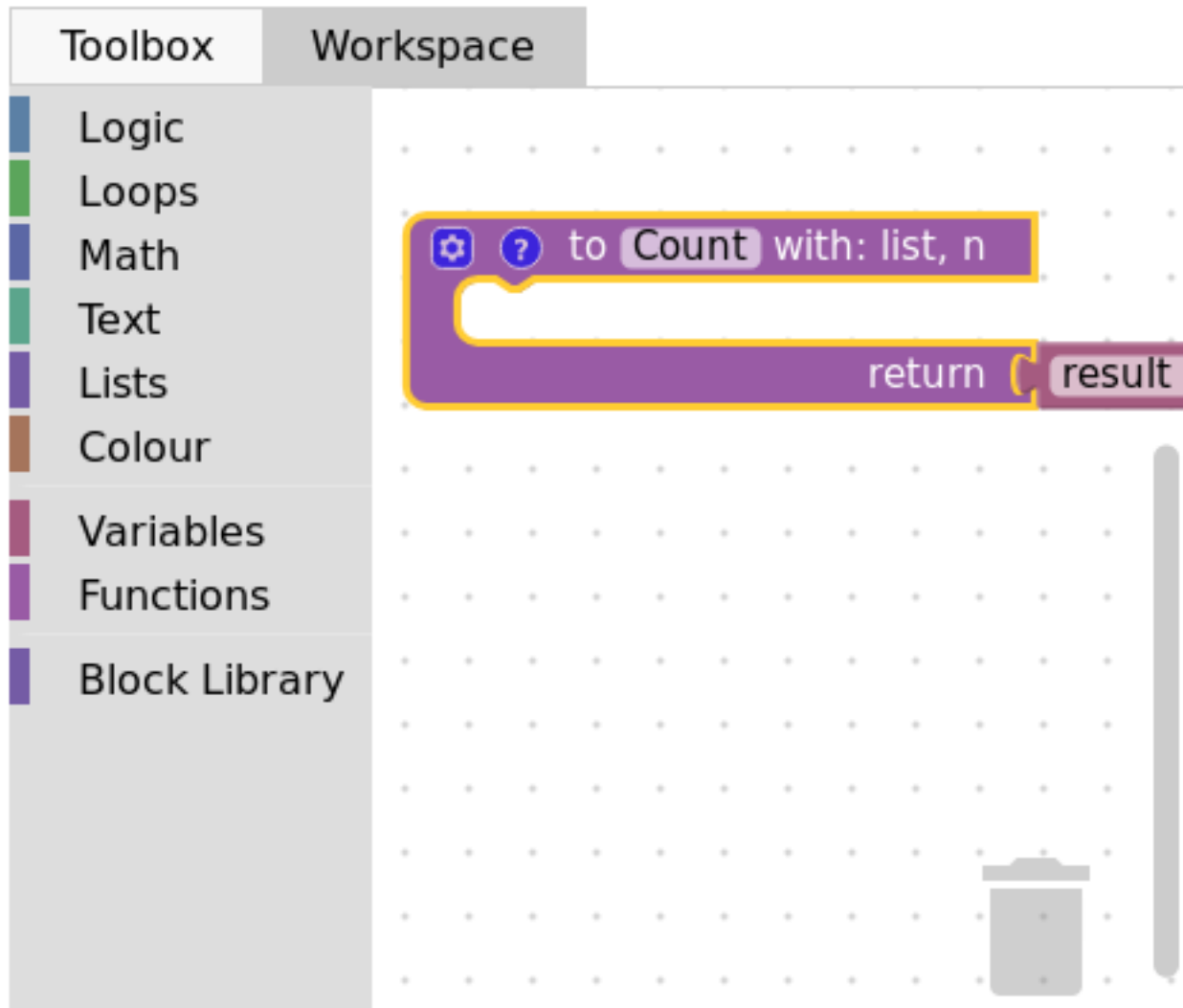


To get the full documentation about what can be achieved when modifying the toolbox manually, head to [this link](#) (Google documentation).

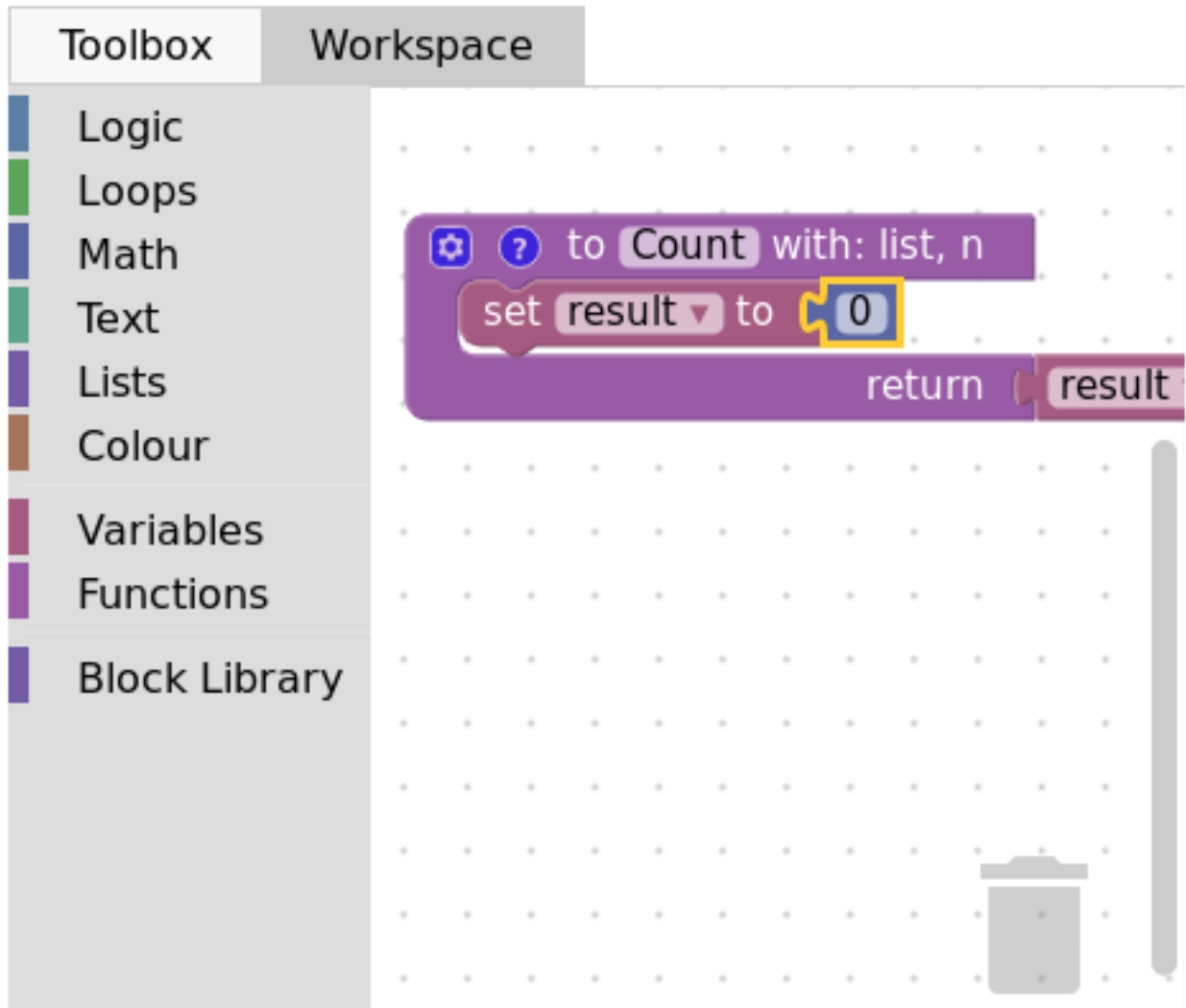
### 1.3 Example : an “only workspace” task

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.

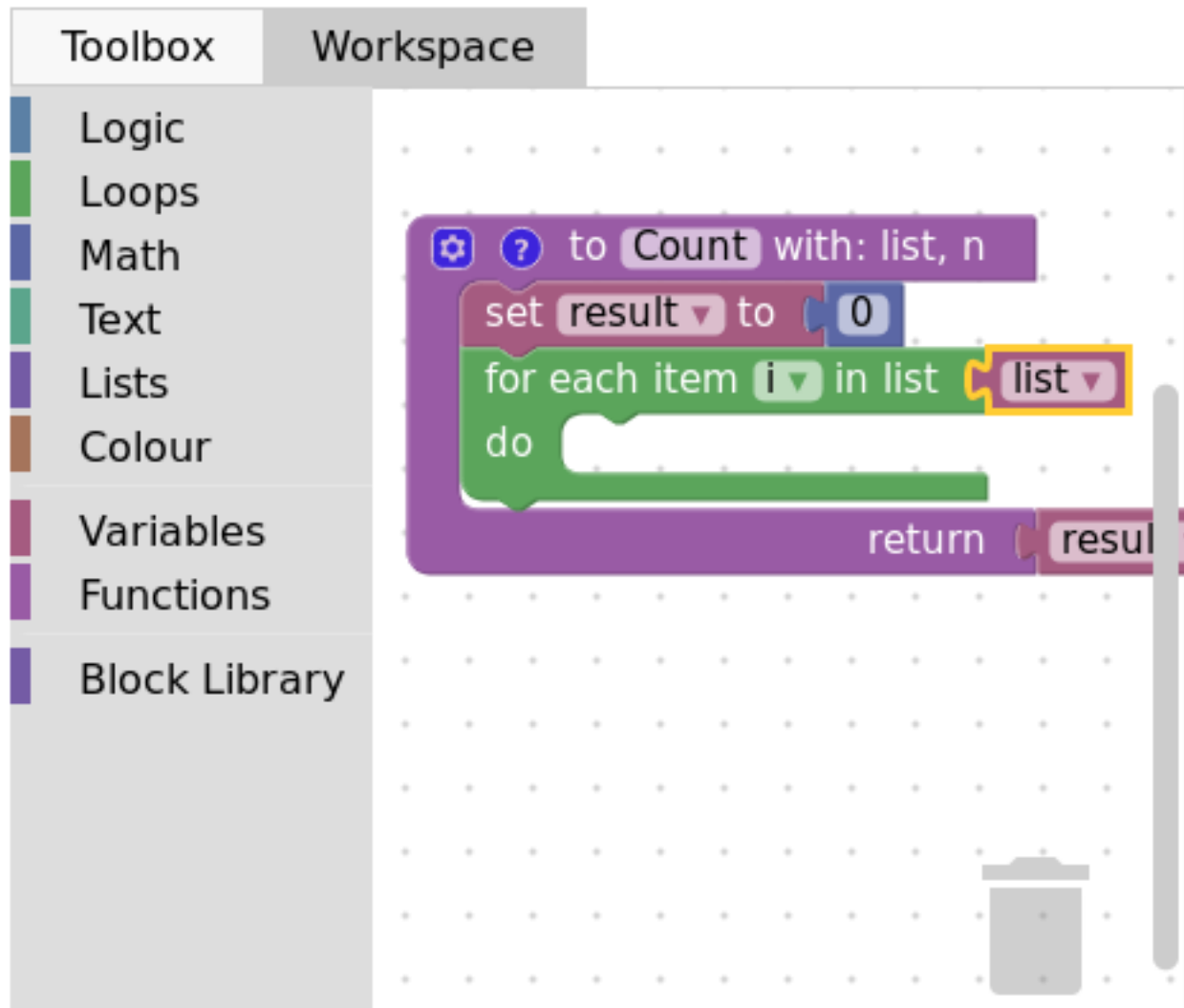
First, open the graphical editor, click on the workspace tab and create a function that takes two parameters *list* and *n*, and returns a value *return* (if you are not familiar with the graphical interface use, refer to [Example : create the sum function \(using the graphical interface\)](#))



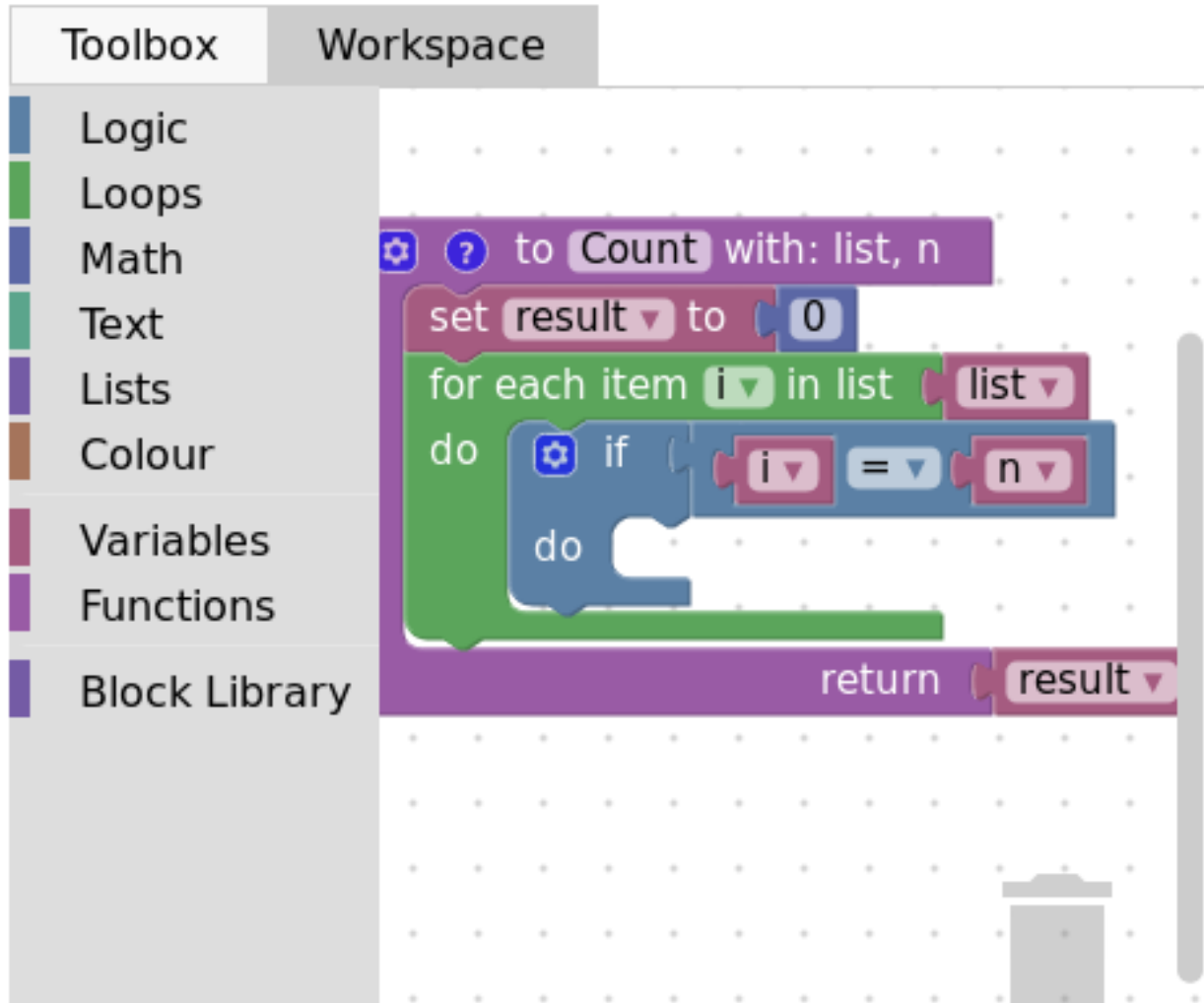
Then, from the *Variables* category, take the “set result to” block, and set it as the first block in the body of the function. From the *Math* category, get the “0” block, to first set result to zero. Here is the current progress :



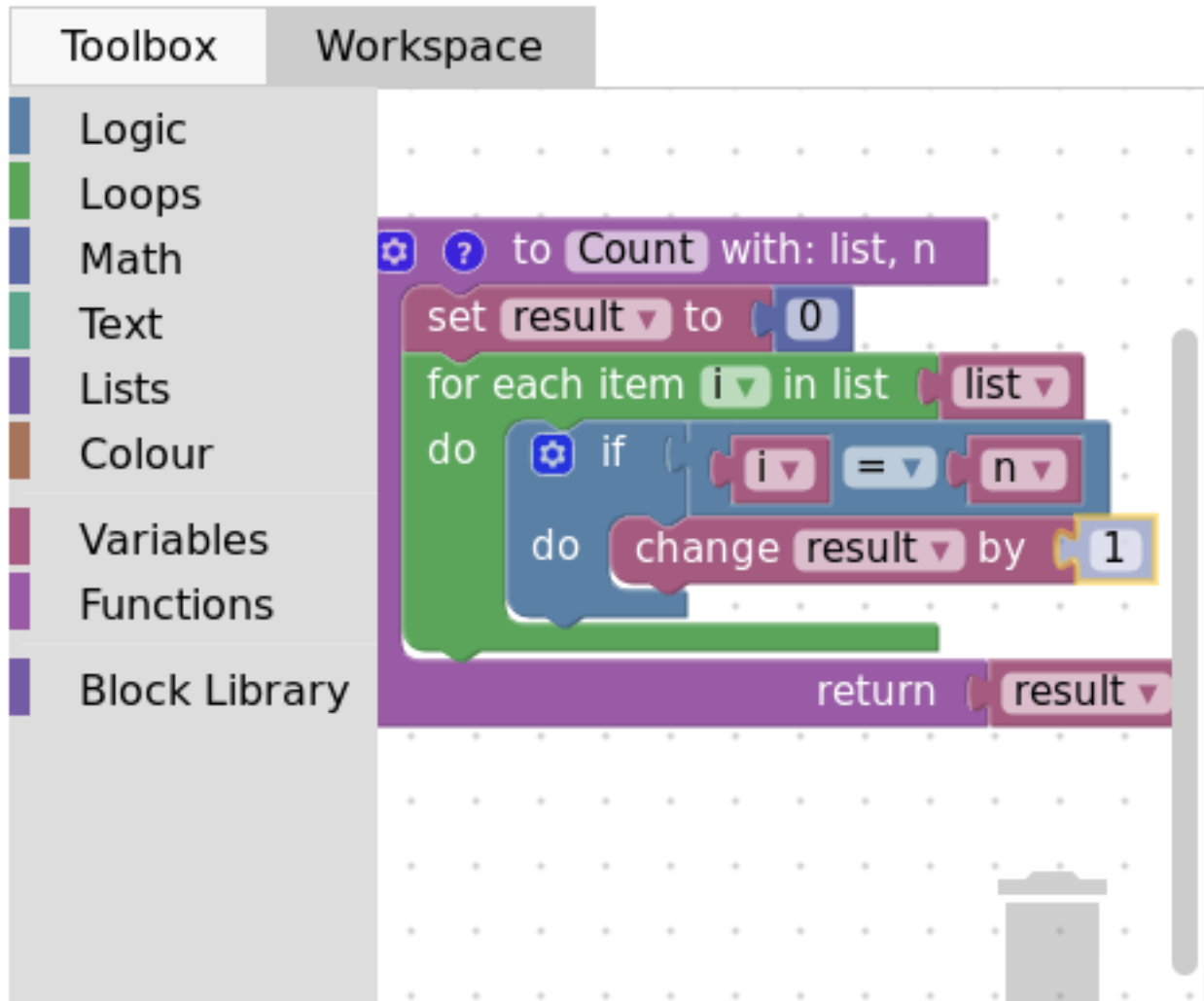
Next, from the *Loops* category, get the “for each item in list” block, plug it under the last one, and get the *list* variable to add it into the bloc :



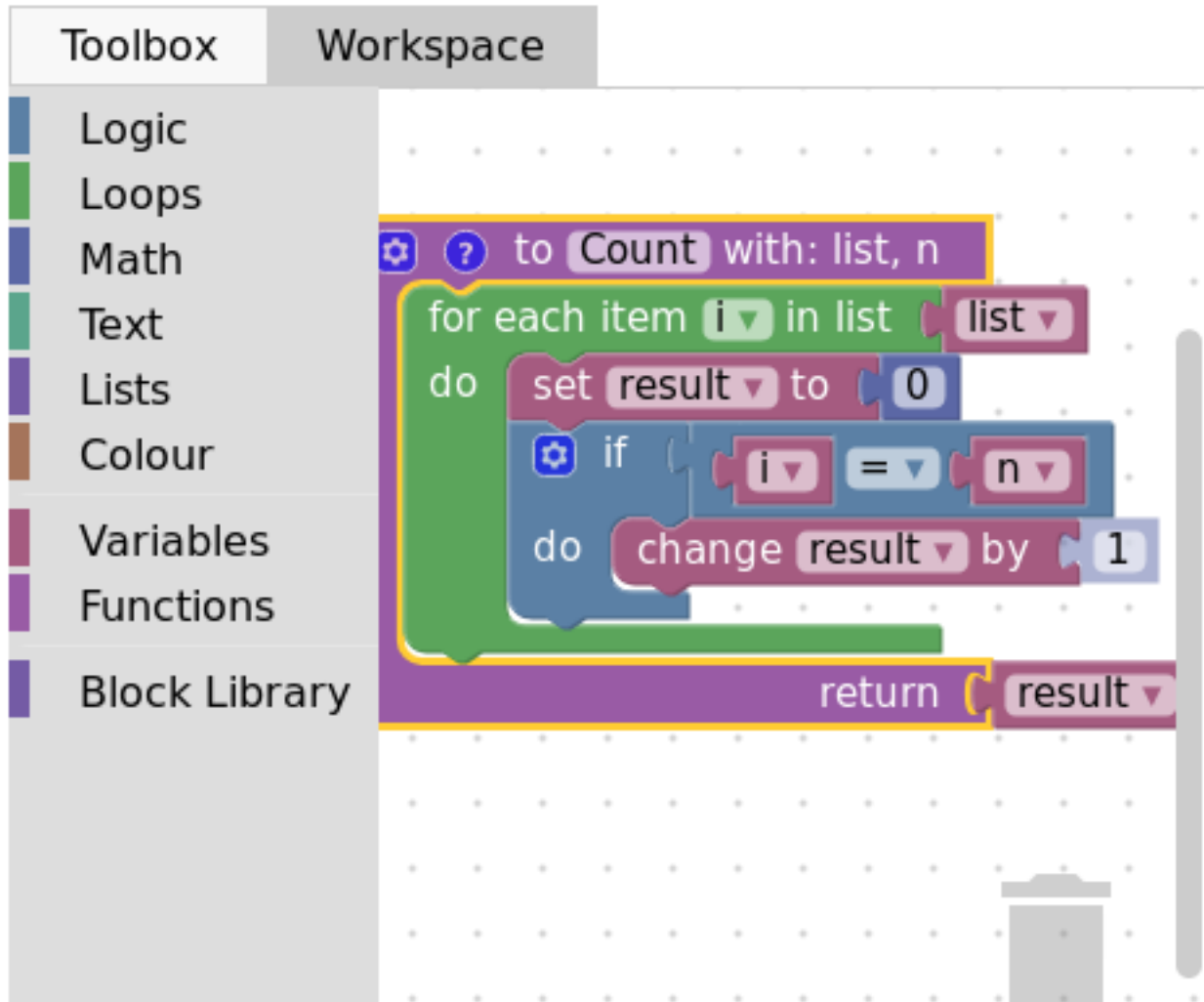
Add the “if” condition from the *Logic* category, and create our boolean  $i == n$  with blocks from *Logic* and *Variables*



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 :

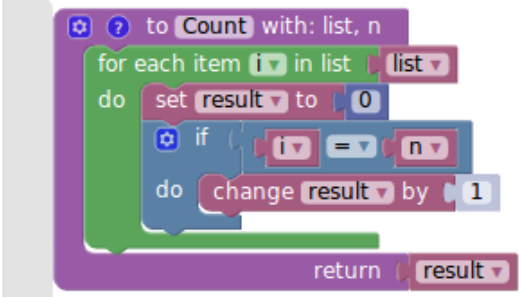


Here is what the student will see on INGINious :

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
```

(continues on next page)



(continued from previous page)

```

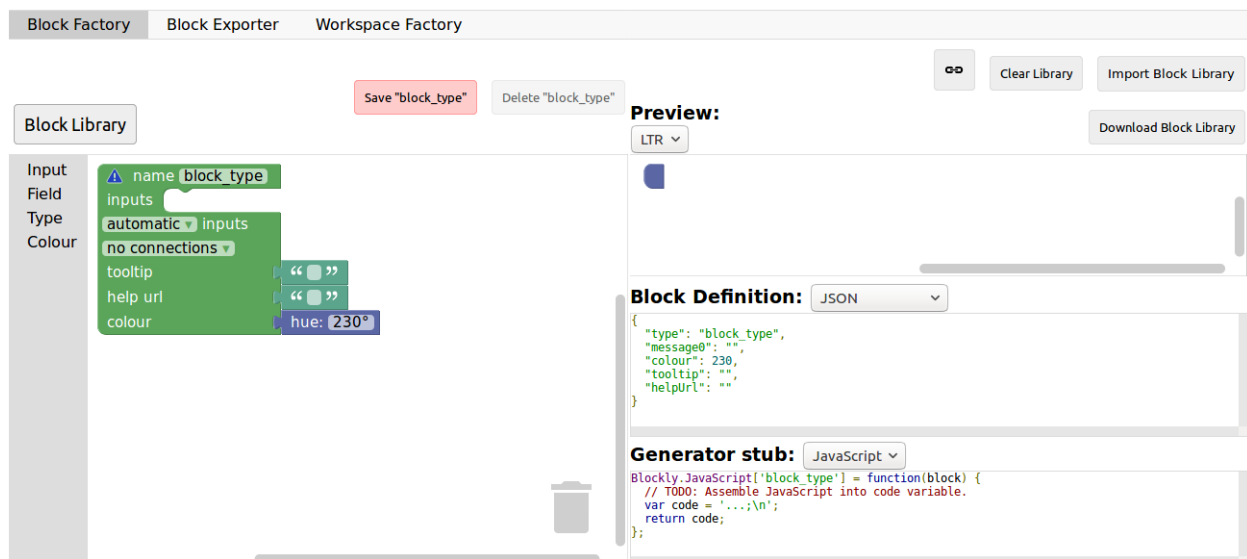
List = []
for j in range(15): #lists of 15 elements
    List.append(random.randint(0,10))
n = random.randint(0,10)
correct = countList(List, n)
output = Count(List, n)
if(correct != output):
    print("For the list "+str(List)+ " and the number "+str(n)+ " you have_
↪returned "
    + str(output) + "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

```

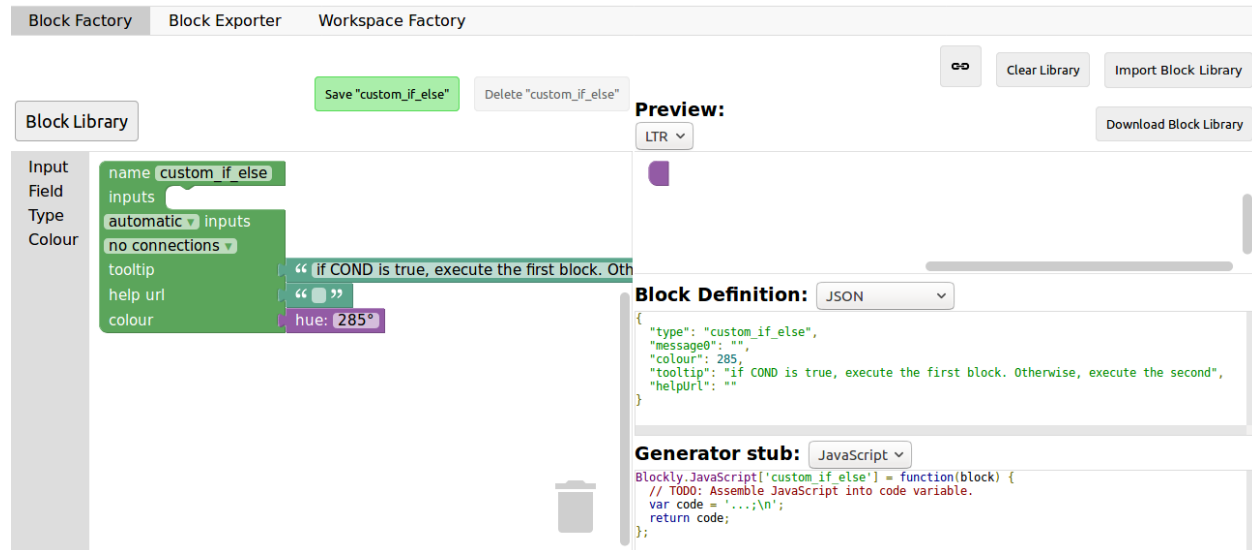
To make the correction and feedback easier, we defined a function giving the correct answer, and compare this function's result the the student one. We then run a few tests on random inputs. With the basic run file and this one in your task folder, it is complete.

## 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 :

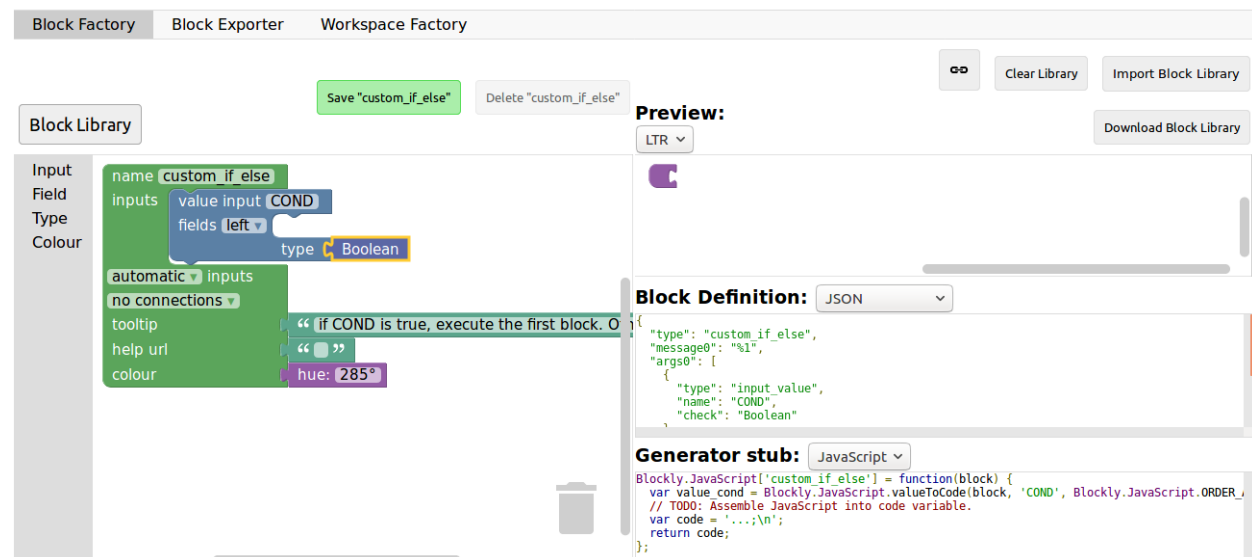


You will construct your block using the left side, while the right side is a live preview of both the visual and the code that will be generated. Let's construct an `if else` block. First, enter a name for it in the top field. It has to be unique across all Blockly blocks, so we will call it `custom_if_else`. 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).

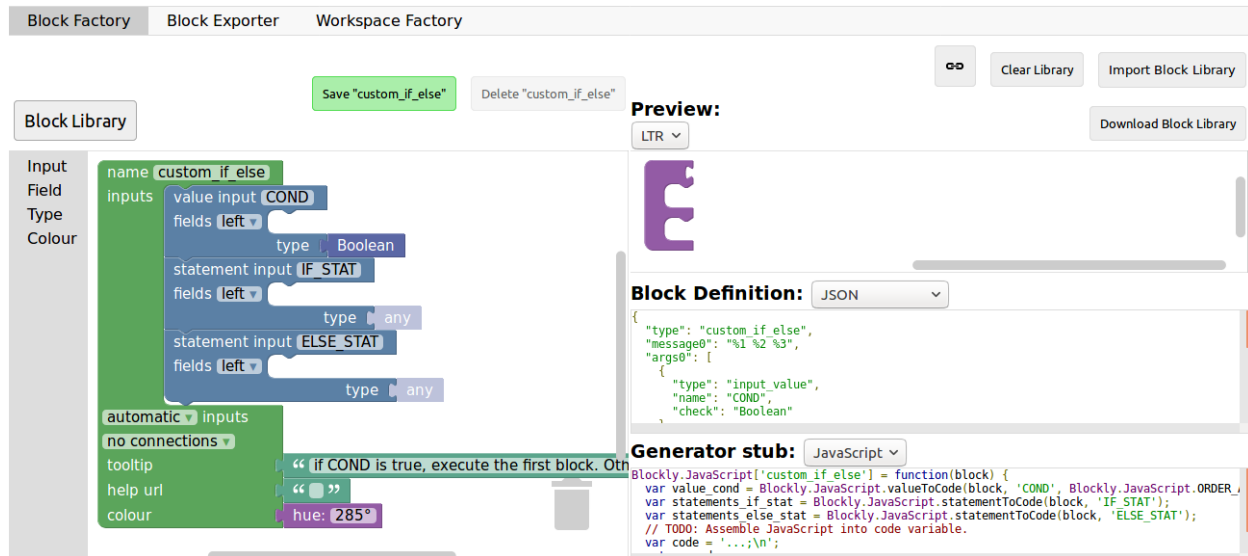


We will now construct the slots that our new block need. Since we are doing an `if else` we need to attach one boolean condition (the `if` condition), and two slots to put statements. This can be done with the *Input* category of the factory. There is three types of inputs : value, statement and dummy.

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*.

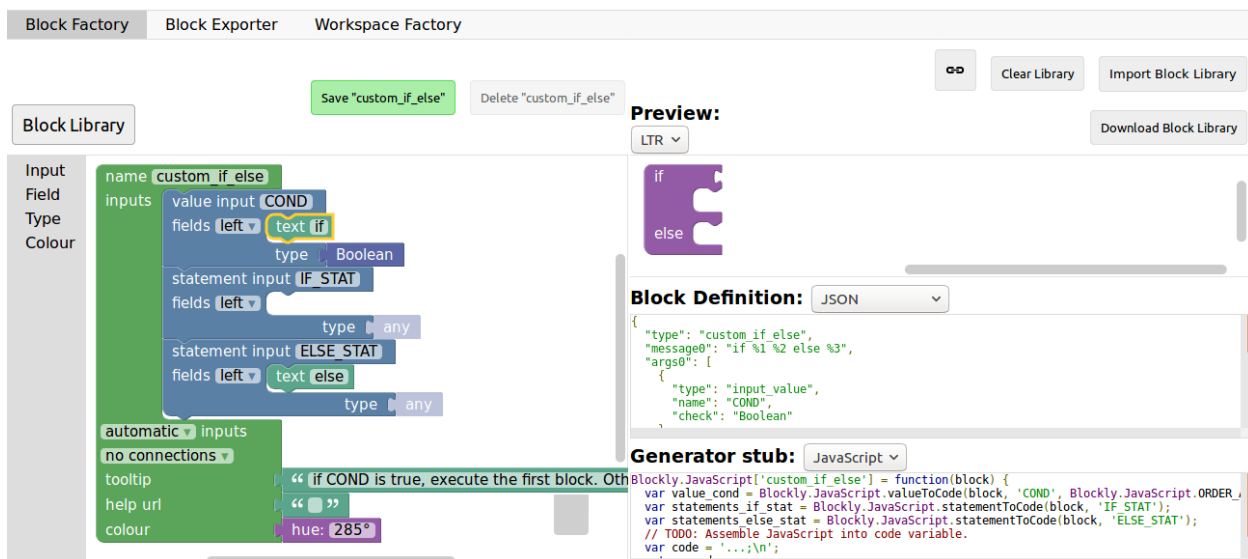


Now, we need the slots to put the statements. Again, click on the *Input* category and drag two *statements* blocks (dummy input won't be used in this tutorial, they simply allow to add extra space to a block for annotations but are not interactive). We need to name those inputs, respectively “IF\_STAT”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

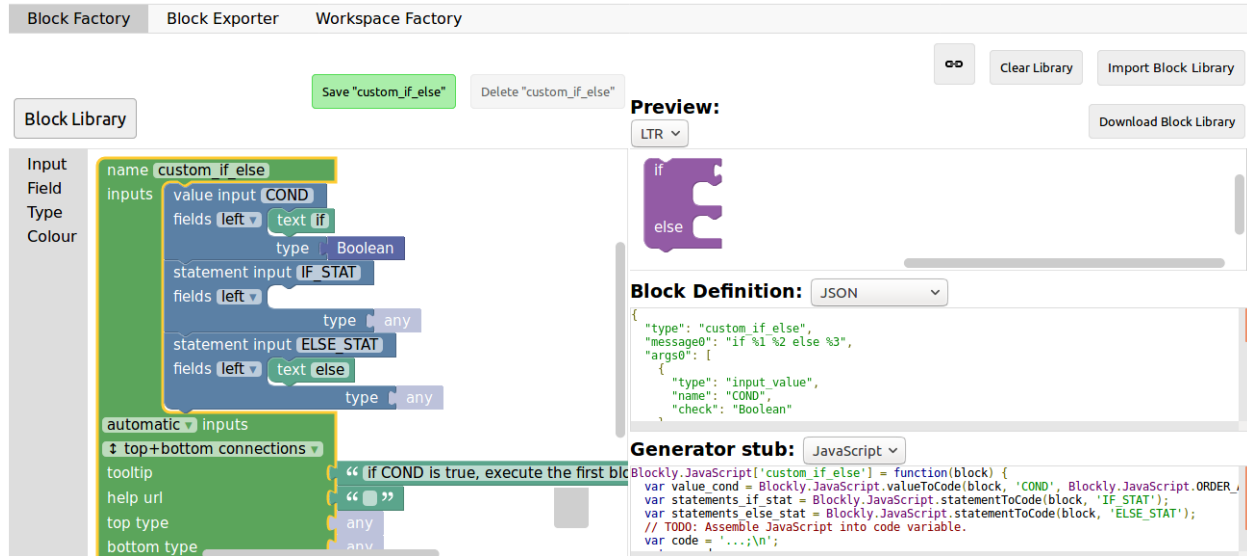


Now, our block has the correct structure, but adding text to it would make it clearer. This can be done using the *Field* category. There is a lot of different field items (user input, drop down, color pickers, ...), to which you can find documentation [here](#).

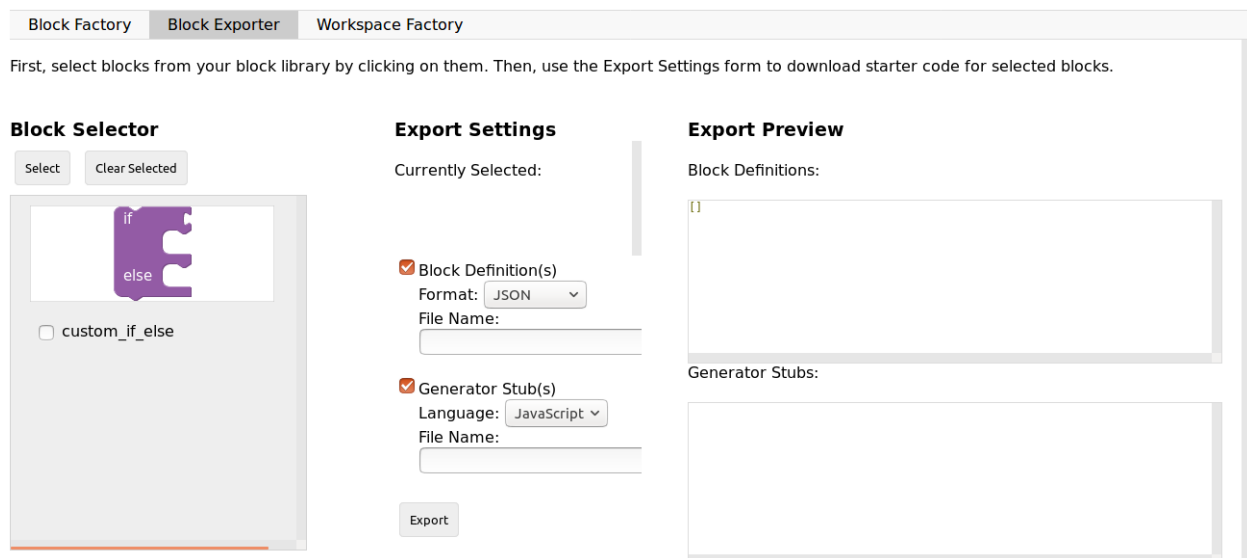
In our case, we need two *text* fields, one in the value input, and one in the second statement input. In the first field, we write "if", and in the second "else" (here, there is no need for the values to be unique).



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). For the generator, we need the Python version of the code, so change the language using the dropdown. For the definition, either Javascript or JSON works, it just has to be integrated differently. 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'] = {
  //JSON or javascript code for the bloc
};

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

For the first function, which is the block description, you can use the javascript code as it has been generated, or put the JSON into this format :

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

In that case, don't forget to remove the extra `[{}]` that surround the json description, as shown in the next snippet of code. Using our generated files, we get :

```
//License
'use strict';

Blockly.Blocks['block_name'] = {
  init: function() {
    this.jsonInit({
      "type": "custom_if_else",
```

(continues on next page)

(continued from previous page)

```

    "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"
      }
    ],
    "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;
};

```

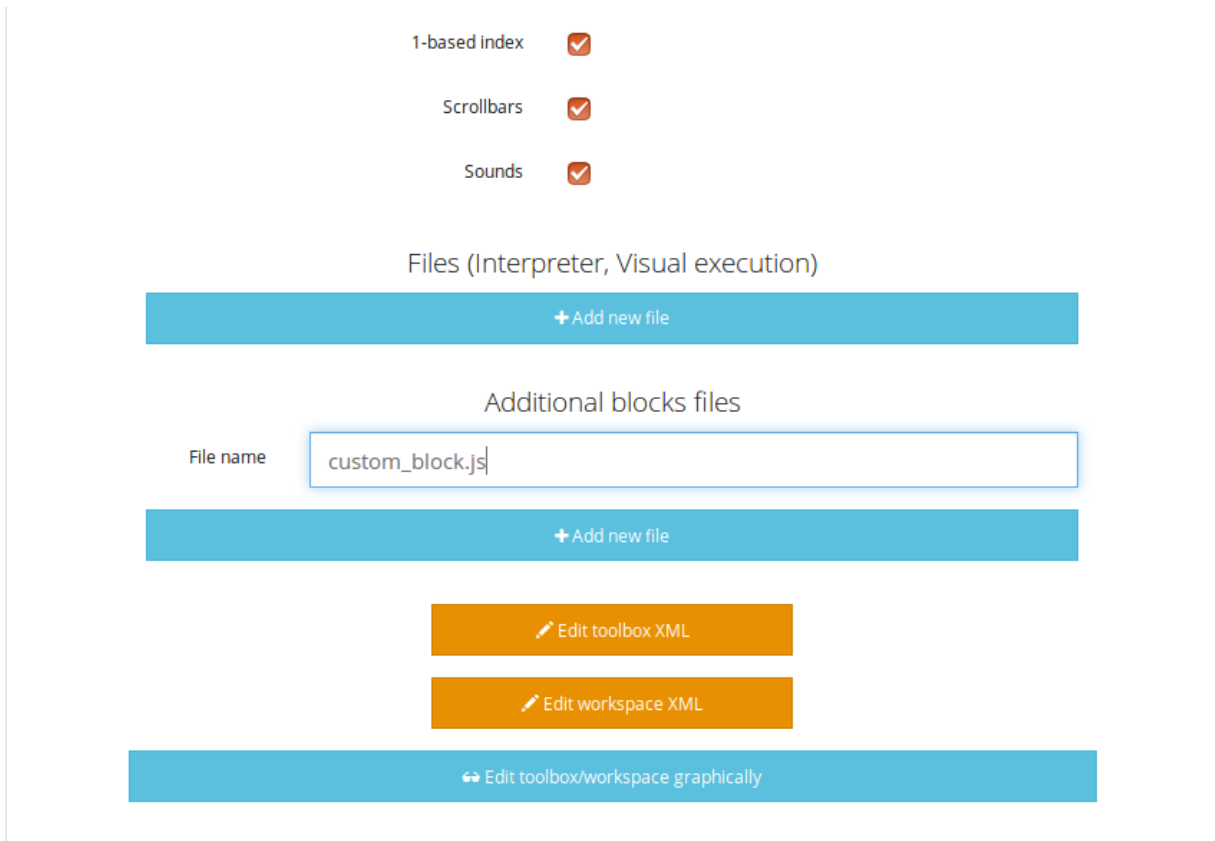
Now, we only need to link all the parts of our block into the corresponding python code. More details on how to get the code out of a block can be found on [this link](#). Here, we simply need to write the if/else structure around the part we already got in the variables and put it in a string :

```

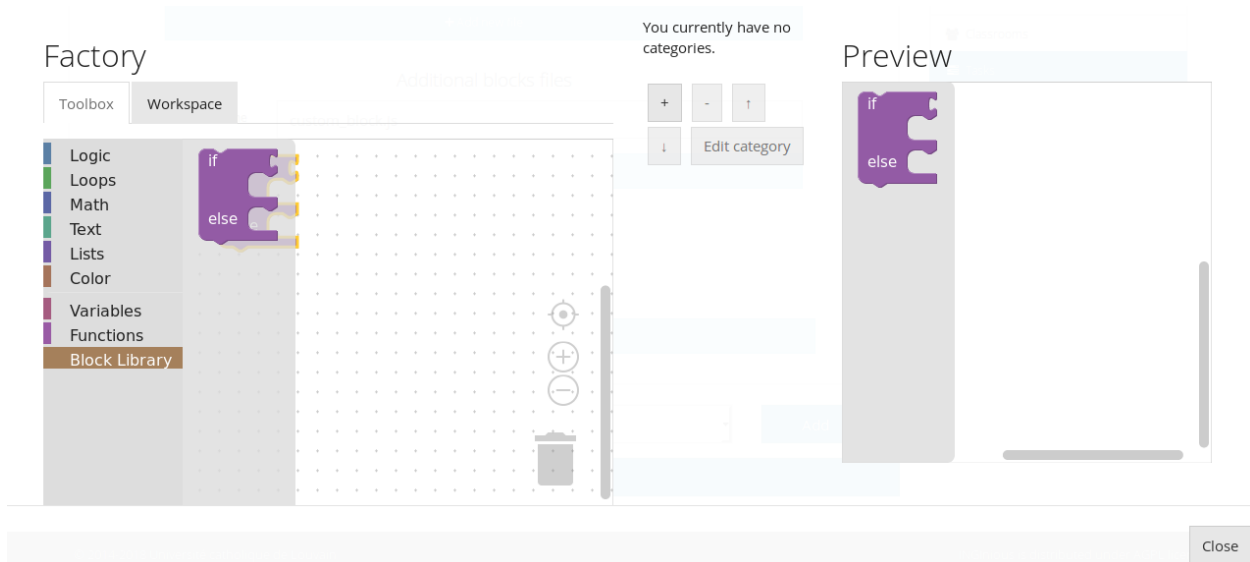
Blockly.Python['block_name'] = function(block) {
  var value_cond = Blockly.Python.valueToCode(block, 'COND', Blockly.Python.ORDER_
↪ATOMIC);
  var statements_if_stat = Blockly.Python.statementToCode(block, 'IF_STAT');
  var statements_else_stat = Blockly.Python.statementToCode(block, 'ELSE_STAT');
  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.



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 :







---

### Blockly visual tasks

---

## 2.1 Available types of tasks

We created a few types of tasks that include a visual and an animations and work with the Blockly plugin. This section will describe each of them and how to use them.

### 2.1.1 Maze task

The first type of task is a maze, where the student needs to go from a point A to a point B, following the allowed paths and eventually avoiding ennemies. Currently, two types of visuals are available for this task : pegman and zombie.

### 2.1.2 Collect and create task

The second type of task has the general layout of a maze, but the goal is to collect and/or create all items present on the map.

### 2.1.3 Drawing task

The final type of visual task require the student to draw shapes on a canvas, according to the instructions.

### 2.1.4 Create a new instance of a task

The easiest way to create a new task of any of the described type is to start from a very basic already existing task, and modify the files according to what you want. Zipped examples can be found at [this link](#). We will describe the conventions for each type of task 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. . .

## 2.2 Create a new maze task

First, download the file `task_pvz`, `task_pegman` or `task_4_enemies`, unzip it to create a task and make any changes you want to the titles, instructions, etc. . .

The only file you have to edit to use this kind of task is `maze_config.json` 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 go over every element of the json file.

Here, we will only look at the `map` element of the file, since the visual part is covered by the other documentation. The first element is `layout`, which is an array of arrays. Each array will represent a map that the student has to solve in this exercise. In our example, we have :

```
{
  "layout": [
    [ [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, 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, 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, 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] ]
  ]
}
```

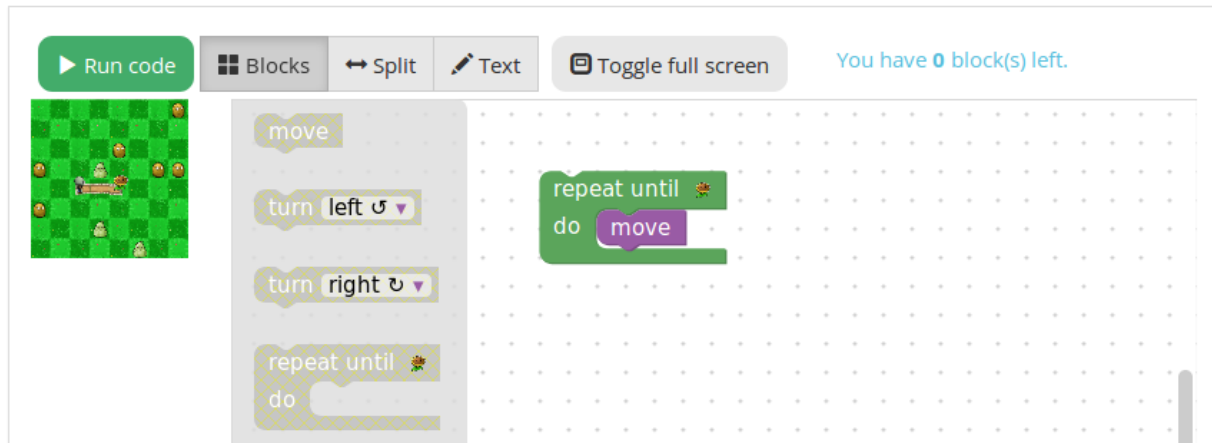
The meaning of the numbers can be found in the item `squareType`, that you should not have to modify except if you are adding new types of elements to the maze. We will go over them in more details here :

- 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

Our two maps will, of course, not be displayed at the same time. They are there to ensure that the student doesn't hardcode the solution to the map he sees. You can add as much maps that you want, with a minimum of one.

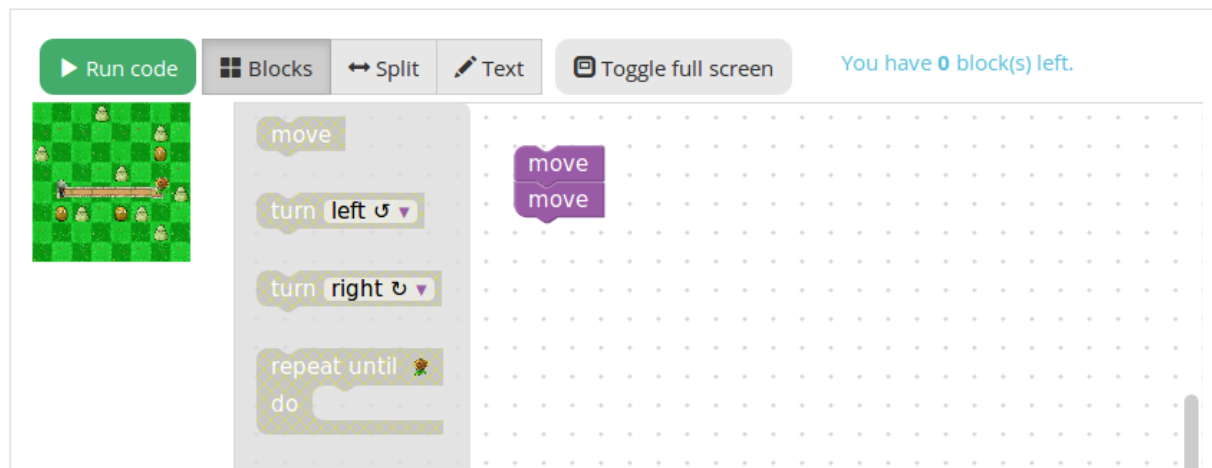
When a student submit his code, the correction will test each map individually (in order). If they all pass, the student succeeded. In our example, it will look like this :

Your answer passed the tests! Your score is 100.0%. [Submission #5b4db59b2726450d1dc3b956]  
Vous avez résolu l'exercice.



If some instance failed, a negative feedback is given and the failed instance is displayed :

There are some errors in your answer. Your score is 0.0%. [Submission #5b4db6172726450d1dc3b960]  
You haven't solved this instance



Let's, for example, modify the first map to have two empty tiles, then a turn with an obstacle and the goal after the straight line. Modify layout to this :

```
{
  "layout": [
    [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, 4, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]]
}
```

(continues on next page)

(continued from previous page)

```
[ [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, 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]]  
}
```

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



If you wish to change the orientation of the character at the start, change the `startDirection` item to one of the values *EAST* (the value our default file use), *SOUTH*, *WEST* or *SOUTH*. For example, here is the output with `"startDirection": "WEST"` :



If you feel like the animation speed is a little bit too quick or too slow, update the value of the `animationSpeed` item.

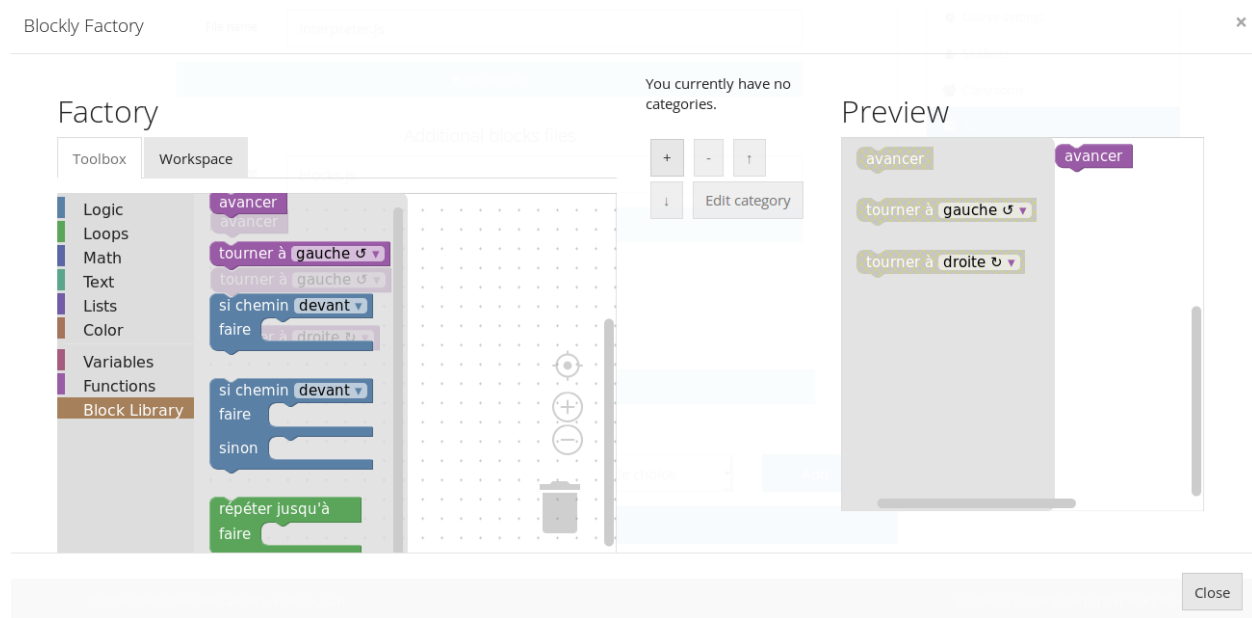
If you wish to change the subproblem id (not visible to user, so we would advise to not do it), you will have to modify the line 107 in the file `yourtask/student/maze.tpl.py` to correspond the name you picked. By default, we have :

```
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 it, 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.



If you are creating a task of type *APP0*, they work as described above, except that there is no path, and walls of some sort replace them as an indication of where the character needs to go. There also is a few more blocs than in the simpler maze used in Code.org.

## 2.3 Create a new collect/create task

First, download the file `task_bee`, unzip it to create a task and make any changes you want to the titles, instructions, etc...

Same as the maze task, the only file you have to edit for this type of task is `maze_config.json` under `yourTask/public`. If you wish to change the graphical look of the task, head to [this part](#) of the documentation to learn how to do so. If not, we will go over every element of the json file.

Again, we will only look at the `map` element of the file. The first element is `layout`, which is a double array (there is no multiple instances correction in this kind of task). As in the previous task, it represents the map, with `squareType` giving the details about the number. Since there is no goal point in this kind of task, the only types are :

- 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

In our example, we have this map :

```
{
  "layout": [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 1, 1, 1, 1, 0, 0],
    [0, 0, 2, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 0, 0],
    [0, 0, 1, 1, 1, 1, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
  ]
}
```

The next item is the dictionary `specialCells`; which describes the elements to collect and create, as well as a special one to hide the spot. It has four entries which are arrays of dictionaries : *honey*, *redFlower*, *purpleFlower* and *cloud*; each containing one type of special element.

In our example, we have one honey, one red flower, one purple flower, one cloud and one element that is hidden underneath the cloud :



Each element has it's own characteristics, which will be detailed underneath

### 2.3.1 Honey or redFlower

The honey is the element that the student will have to create, and the red flower is one of the element to collect.

Here is what is in the `specialCells` element for honey and redFlower :

```
{
  "honey": [
    {
      "x": 4,
      "y": 4,
      "value": 2
    }
  ],
  "redFlower": [
    {
      "x": 3,
      "y": 4,
      "value": 2
    }
  ],
}
```

(continues on next page)

(continued from previous page)

```

        {
          "x": 3,
          "y": 3,
          "value": 2,
          "or": "honey"
        }
      ]
    }
  }
}

```

Each of those elements must have an `x` and `y` coordinates that will place them on the map, as well as a `value` representing how many time they must be collected or created (on the visual, it can be seen as the number on the right corner). Then, there is two optional arguments :

- `or` followed by the name of the other element (either *honey* or *redFlower*), makes it so, when the map is generated by charging the page, this element will either be of it's type or the other one (a red flower or a honey).
- `optional` followed by the boolean `true`, makes it so, when the map is generated, the element is present or not (at random)<sup>1</sup>.

### 2.3.2 PurpleFlower

The purple flower is the other element to collect, its value is in a specified range, and is denoted with a `?` before the animation starts.

The `specialCells` content is :

```

{
  "purpleFlower": [
    {
      "x": 5,
      "y": 5,
      "range": [0, 2]
    }
  ]
}

```

This element also has the `x` and `y` coordinates, but no value. Instead, it has the item `range`, which specify that it's value will be an integer between 0 and 2 (included) in our example. It also has one non-required argument :

- `optional` followed by the boolean `true`, makes it so, when the map is generated, the element is present or not (at random)<sup>1</sup>.

### 2.3.3 Cloud

This last element is used to hide parts of the map before the start of the animation (it will desappear as soon as the user clicks "run code").

The `specialCells` content is :

```

{
  "cloud": [
    {
      "x": 3,

```

(continues on next page)

<sup>1</sup> This argument is common for all elements



(continued from previous page)

```

        "y": 3
    }
]
}

```

This element only has the `x` and `y` coordinates, as well as the optional argument<sup>1</sup>.

As for the maze, if you feel like the animation speed is a little bit too quick or too slow, update the value of the `animationSpeed` item.

If you wish to change the subproblem id (not visible to user, so we would advise to not do it), you will have to modify the line 138 in the file `yourtask/student/maze.tpl.py` to correspond the name you picked. 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@@

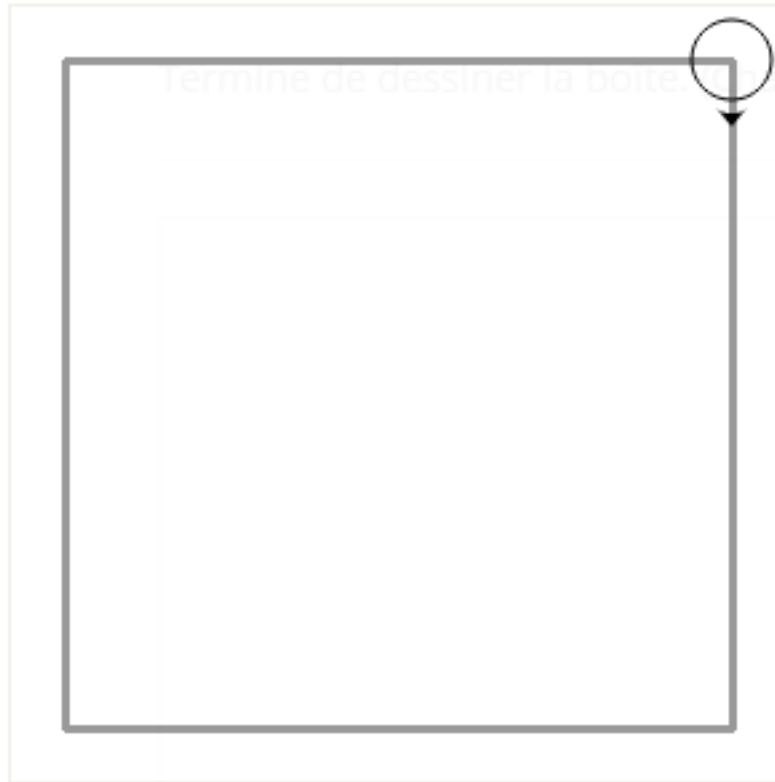
```

The task should now work as expected. If you wish to add or remove blocks from it, 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.

## 2.4 Create a new drawing task

First, download the file `task_artist`, unzip it to create a task and make any changes you want to the titles, instructions, etc...

Here is what our example looks like in INGIous. The circle with the arrow is the drawing utencil, is is referenced further down as “the turtle” it moves (and draw) in the direction of the arrow.



For this kind of task, you will have to edit a few files (and eventually run a Python script). Let's first go over the json configuration file (under `yourTask/public`), `turtle_config.json`. Here is the content of our example :

```
{
  "startX":270,
  "startY":20,
  "startAngle":180,
  "strokeWidth":3,
  "strokeColour":"#000000",
  "colourSpecific":false,
  "radius":15,
  "animationRate":50,
  "width":290,
  "height":290,
  "imageSolution":true,
  "imageName":"solution.png"
}
```

Let's detail each item :

- `startX` is the X coordinate on which the turtle will start. (0, 0) is the top left corner.
- `startY` is the Y coordinate on which the turtle will start.
- `startAngle` is the starting angle (0 is facing pure north, 180 pure south)
- `strokeWidth` is the width of the lines that will be drawn
- `strokeColour` is the start color of the turtle (the student can modify it)
- `colourSpecific` must be a boolean, and is true if the color of the line matters when correcting the exercice
- `radius` defines the radius of the turtle body

- `animationRate` defines the time (in ms) between frames
- `width` is the width of the canvas
- `height` is the height of the canvas
- `imageSolution` must be a boolean specifying if the solution to be drawn on the canvas is an image or not (true if it is)
- `imageName` not used if `imageSolution` is false, specify the name of the solution image

The second step is to define how the expected solution will be displayed to the user. You can either create the corresponding image using the script `create_img.py` or write the code in the `turtle.js` file. We will describe both methods.

### 2.4.1 Create solution image using *create\_img.py*

To use this script, you must have the Python module Pillow installed (see [this link](#)).

Then, write the solution to your exercise in the file `create_img.py` (from the line 95). The function names used by Blockly are available, so you can solve your exercise using blocks and copy/paste if you feel that this is easier for you<sup>2</sup>. In our example, we have this :

```
for i in range(4):
    moveForward(250)
    turnRight(90)
```

Finally, in your public directory, run `python create_img.py`. The script creates your solution image named *solution.png* (you can rename it as long as you don't forget to change the name in the config file), and it will be used as long as the `imageSolution` boolean in the `turtle_config.json` file is set at true.

### 2.4.2 Write the solution code in *turtle.js*

If you pick that option, you must be aware that the solution to the exercise will be available to a student clever enough to find the correct Javascript file and read it carefully.

Open the file `turtle.js` in the public directory, and edit the function `solution` (line 24) to execute your solution using the Javascript versions of the functions used by Blockly (they are all in the form `Turtle.function()`). In our case, we have :

```
var solution = function(){
    for(var i = 0; i < 4; i++){
        Turtle.moveForward(250);
        Turtle.turnRight(90);
    }
}
```

Save the file, and this code will be used as long as the `imageSolution` boolean in the `turtle_config.json` file is set at false.

### 2.4.3 Write solution in the correction file

The correction file, `turtle.py` (under the `student` directory), also need to have the correct code for the task. If you have picked the option of using the `create_img.py` script, you can simply copy/paste your code from line 21.

<sup>2</sup> You will be able to reuse this code for the correction part.

If you have picked the other option, you'll have to translate the Javascript solution code to Python, which is just a matter of translating the loop syntax, removing semicolons and the `Turtle.` part of a function call. Here is what we have in our example :

```
def solution():
    for i in range(4):
        moveForward(250)
        turnRight(90)
```

Now, you should have a working task. Again, if you wish to change the subproblem id (not visible to user, so we would advise to not do it), you will have to modify the line 16 in the file `yourtask/student/turtle.py` to correspond the name you picked. By default, we have :

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

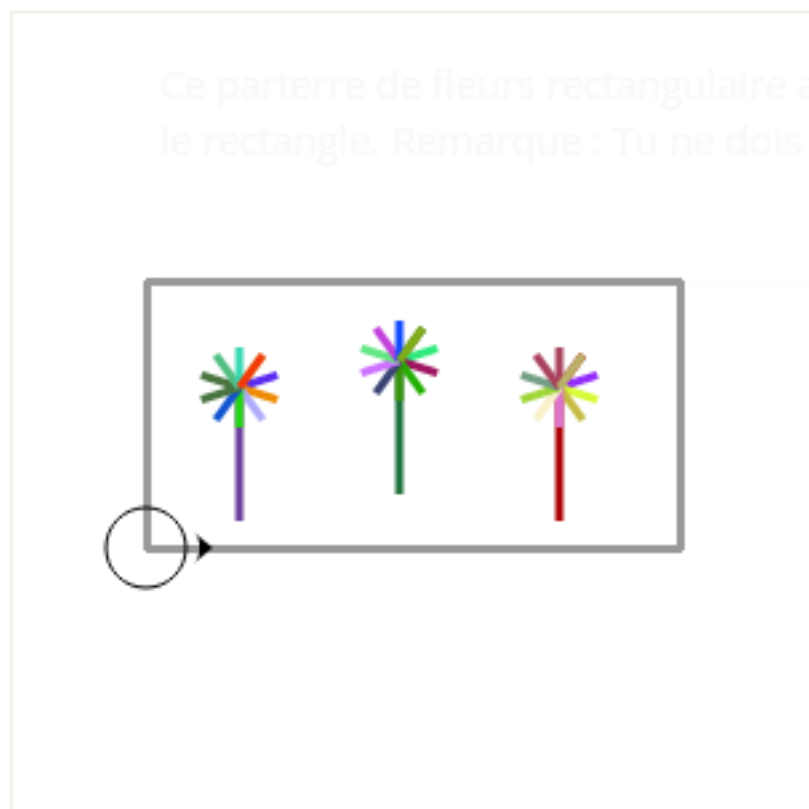
If our subproblem name is *example*, we need to change the value to :

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

Finally, you can also add some decor to the INGIInious display to the student (that will not be considered when correcting). To do so, refer to the next section.

### 2.4.4 Add decoration to the student display

You may want to spice up the display with some decor, like you can see on this image :



To do so, you'll have to edit the `turtle.js` file (under the public directory). Go to the `decoration` function, and add any Javascript version of the Blockly code (eventually creating helper functions). The result will be drawn from the same starting point specified in `turtle_config.json` and will be shown in the task. The code for our example here is :

```
//Code of the decor
var decoration = function(){
    //Here, put the code for any decor, not part of the exercise
    Turtle.penUp();
    Turtle.move(35);
    Turtle.turnLeft(90);
    Turtle.move(10);
    Turtle.penDown();
    drawFlower();
    Turtle.penUp();
    Turtle.turnRight(90);
    Turtle.move(60);
    Turtle.turnLeft(90);
    Turtle.move(10);
    Turtle.penDown();
    drawFlower();
    Turtle.penUp();
    Turtle.turnRight(90);
    Turtle.move(60);
    Turtle.turnRight(90);
    Turtle.move(10);
    Turtle.turnRight(180);
    Turtle.penDown();
    drawFlower();
}

var drawFlower = function(){
    Turtle.penColour(randomColour());
    Turtle.moveForward(50);
    for (var i = 0; i < 11; i++) {
        Turtle.penColour(randomColour());
        Turtle.penDown();
        Turtle.turnRight(36);
        Turtle.moveForward(15);
        Turtle.penUp();
        Turtle.moveBackwards(15);
    }
    Turtle.turnLeft(36);
    Turtle.moveBackwards(50);
}
```



---

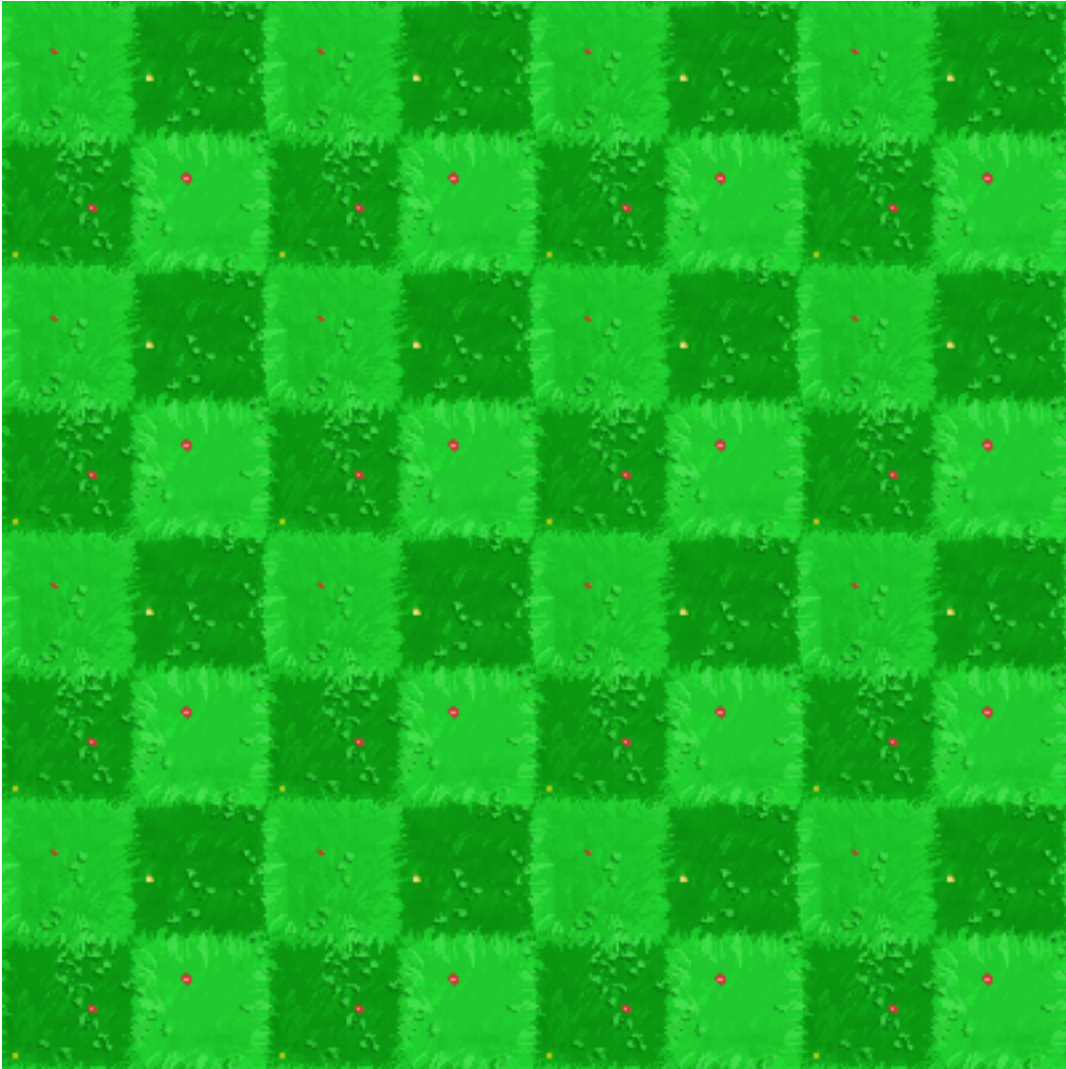
### How to change the maze's visuals

---

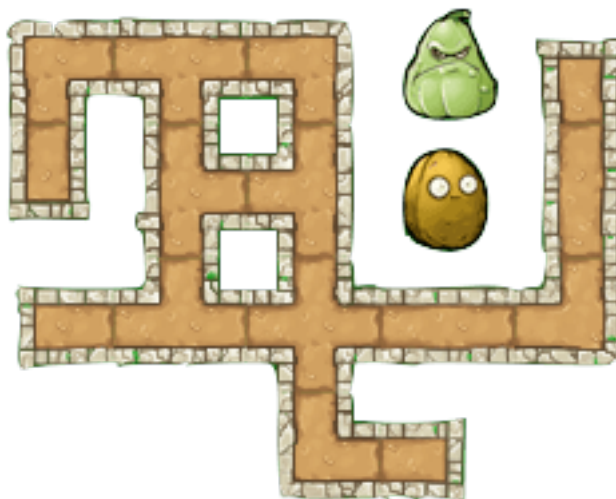
#### 3.1 Files you will need

Depending on the type of task you want to create visuals for, you will need different files. The following ones are common for all of them and have to be 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.



### 3.1.1 Files for a maze task

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)

Now, for the obstacles, there is two ways : either you provide just one type of ennemy that can attack from every “side”, or you want more variety, and can create four ennemies, that are accessed from : the top, the bottom, the right and the left. No matter which you choose, you will need, for each ennemies and “idle” and a “attacking” image. For example, here we have an idle obstacle (when the character is on another tile), in gif or png :

The same 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

### Particular case of Maze : APP0

The type of task *APP0* is a particular case of a maze, where you do not have any animations for ennemies or goal (since the visuals are changing each year). Also, those mazes are made to be any size, so the background to provide is a single tile (a simple texture would work), and will be repeated as much as needed to create the tiles.

Furthermore, there is no path drawn on the map (all free spaces are blank tiles), and there is an illustration for a wall. The json file is the following :

```
{
  "visuals": {
    "sprite": "avatar.png",
    "marker": "nedstark.png",
    "obstacleIdle": "wolf.png",
    "wall": "wall.png",
    "obstacleScale": 1.2,
    "background": "testBackPlain.png",
    "graph": "black"
  },
}
```

### 3.1.2 Files for a collect and create task

An item to collect that will have a fixed value (that do not change when refreshing the page):



An item to collect that have a changing value :



An item to create :



An item to hide some tiles, and a disappearing version of the item (you might need to refresh the page to see the animation):



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

## 3.2 File to modify

The only file to modify is `maze_config.json`, that you will find under `taskname/public`. Here, we only consider the `visual` item (the other one concerns the layout of the task). Here is the common variables :

```
{
  "visuals": {
    "sprite": "avatar.png",
    "tiles": "tiles.png",
    "obstacleScale": 1.7,
    "background": "background.png",
    "graph": false,
    "obstacleSound": [],
    "winSound": [],
    "crashSound": []
  }
}
```

Here, the `background` variable correspond to the 8x8 background, the `tiles` is all of the paths, and the `sprite` variable will hold your avatar image. `obstacleScale` scales items regarding the character, `graph` allows to draw a grid on the map, and the three last variables can hold your sounds if you have them.

To change a file, simply rename your file or change the name in the file, like so :

```
{
  "visuals": {
    "sprite": "myAvatarName.png",
    #Rest of the parameters
  }
}
```

### 3.2.1 Files used by a maze

If you are creating a maze, in `visuals`, you have the following items :

```
{
  "visuals": {
    "marker": "goal.gif",
    "goalAnimation": "goal_win.gif",
    "obstacleIdle": "obstacleIdle.gif",
    "obstacleAnimation": "obstacle.gif"
  }
}
```

The first two contains your end marker as well as a win animation, and are always the same. The next two concern the obstacle, and must be used as described if you have only one ennemy. If you have four, you must define them like so :

```
{
  "visuals": {
    "obstacleIdle": ["obstacleDownIdle.gif", "obstacleLeftIdle.gif", "obstacleUpIdle.
↪ gif", "obstacleRightIdle.gif"],
    "obstacleAnimation": ["obstacleDown.gif", "obstacleLeft.gif", "obstacleUp.gif",
↪ "obstacleRight.gif"]
  }
}
```

The order you put your animations in is very important, and must be like so : obstacle when the character is coming from a tile down it, from a tile to it's left, from up it, from a tile to it's right. Of course, it must correspond to the other version.

### 3.2.2 Files used by collect and create

If you are creating a collect and create task, you have the following additional variables :

```
{  
  "visuals": {  
    "redFlower": "redFlower.png",  
    "purpleFlower": "purpleFlower.png",  
    "honey": "honey.png",  
    "cloud": "cloud.png",  
    "cloudAnimation": "cloud_hide.gif"  
  }  
}
```

The variables correspond to :

- redFlower : the collectable that has a fixed value
- purpleFlower : the collectable that has a variable value
- honey : the creatable item
- cloud : the item used to hide a tile
- cloudAnimation : the animation revealing the tile

## CHAPTER 4

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`