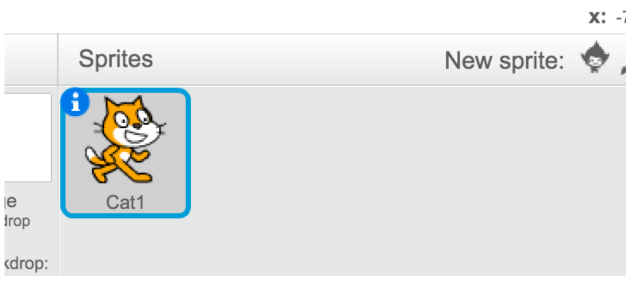
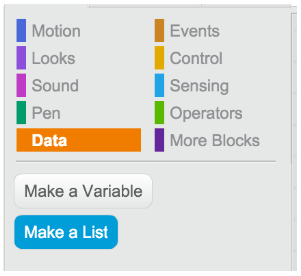
Bubble Sorting Algorithm

Step 1 - Build a List

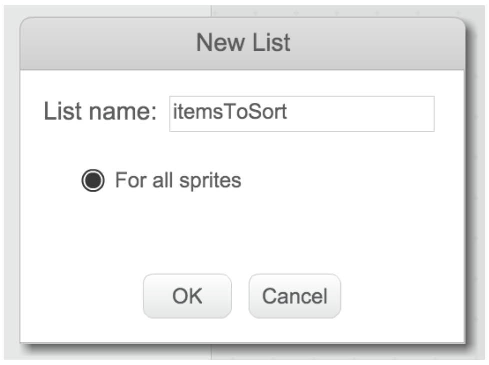
- ☑ Click on the default sprite, the code will be added to the sprite, we can use the sprite to 'say' things later on to indicate that the program is working.



- ☑ Under Data, click on Make a List



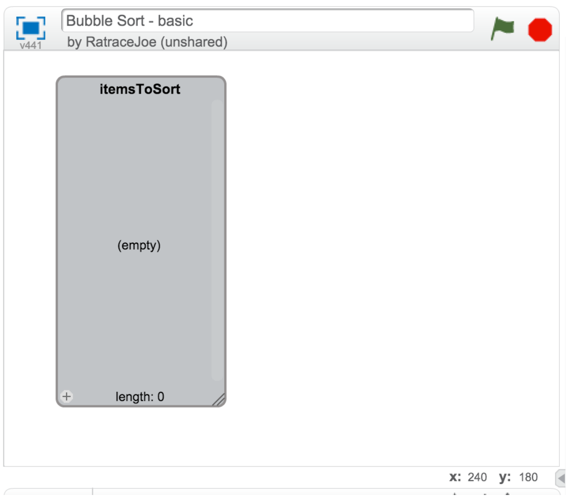
- ☑ Enter list name and click OK. I have used 'itemsToSort'.



List name: itemsToSort

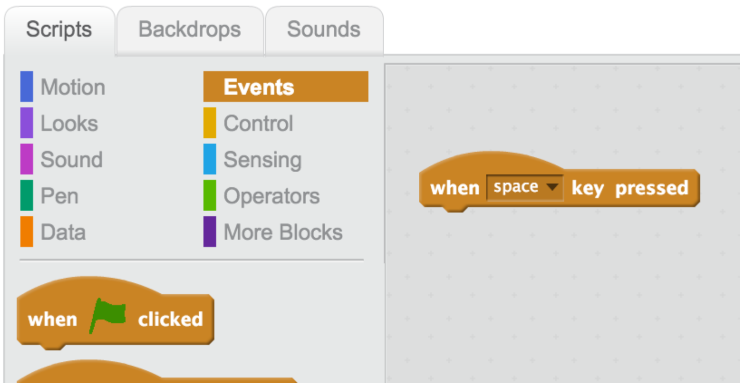
● For all sprites

- ☑ See the list in the game area. This allows you to see the contents of the list as they change.

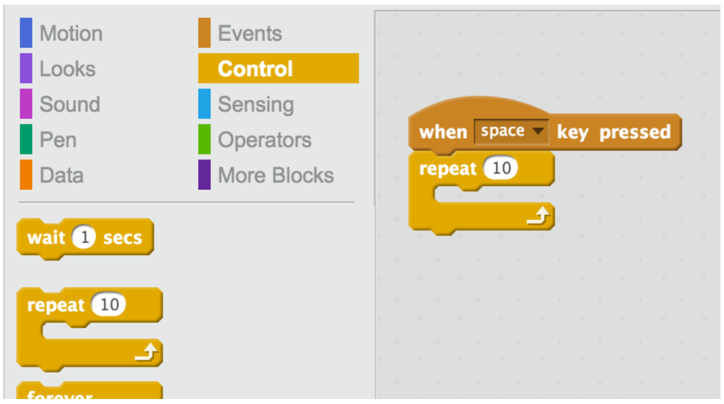


Step 2 - Populate the List on pressing Space Bar

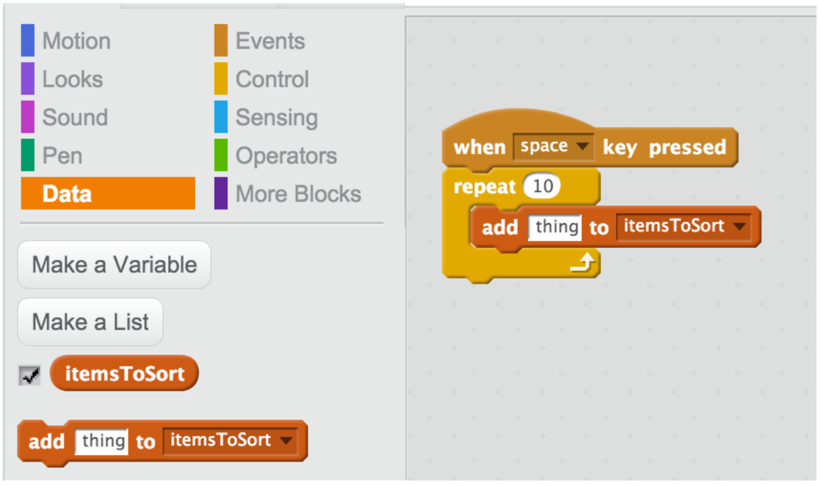
☑ In the Scripts tab, find the Events section, drag a **When space key pressed** block onto the code area.



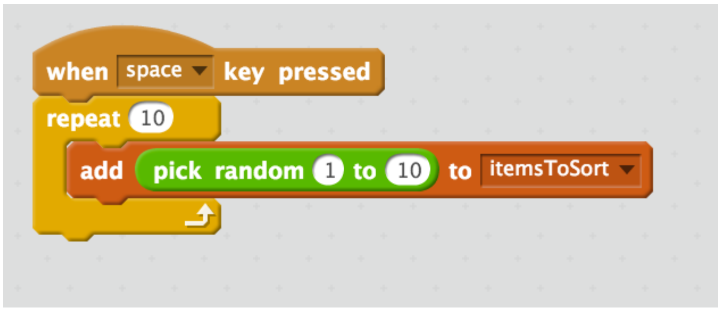
☑ In the Control section, drag **repeat <10>** block under the **When space key pressed** block.



☑ Back under Data, drag a **Add thing to itemsToSort**.

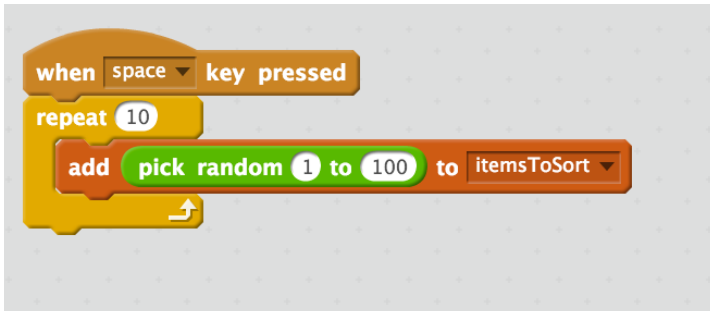


☑ We want to insert a random number, this can be done using the block under Operators. Drag **Pick random 1 to 10** block into the white space in the repeat block. When the white space highlights it means the random generator can be dropped into place.



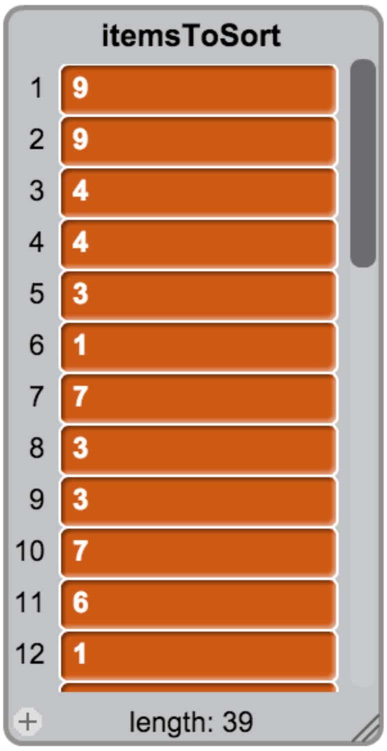
```
when space key pressed
repeat 10
    add pick random 1 to 10 to itemsToSort
```

☑ Change the range of the random generator to **1 to 100.** This will spread the numbers out a bit and make the sorting process clearer.



```
when space key pressed
repeat 10
    add pick random 1 to 100 to itemsToSort
```

☑ Now test it out, click into the game area (this gives it the focus) and then hit space bar a few times.



itemsToSort

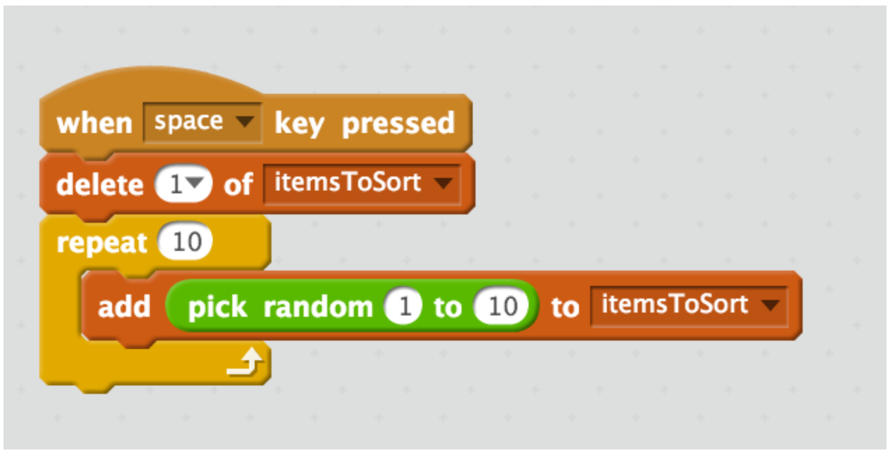
1	9
2	9
3	4
4	4
5	3
6	1
7	7
8	3
9	3
10	7
11	6
12	1

length: 39

Step 3 - Control the List size

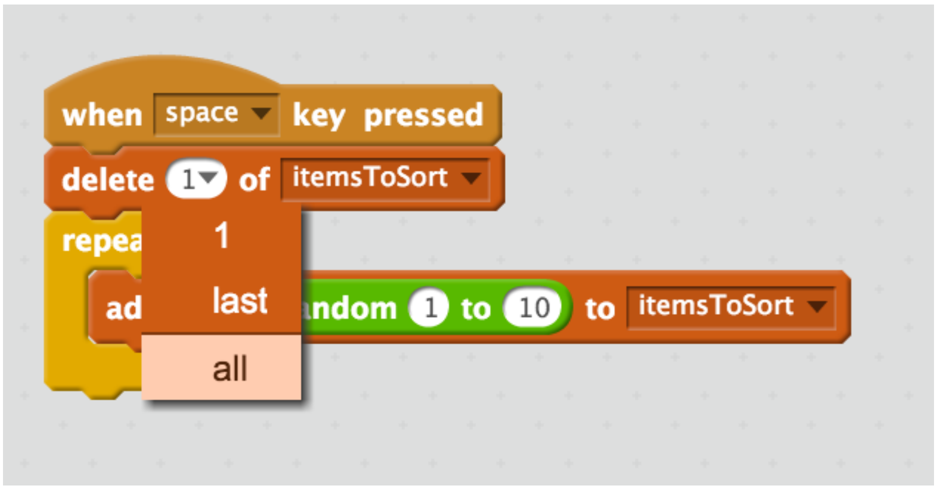
Notice that each hit of the space bar adds 10 more items, so the list continually gets bigger. This is not helpful for demonstrations purposes because the list size means the user has to scroll to see the whole list. It also means that the list size can get out of control. The solution is to tell the computer to clear the list before adding the 10 items. This ensures that there will always be 10 items in the list, but no more.

☑ In the Data section, drag a **Delete <1> of itemsToSort** block. Drag it just underneath **When <space> key pressed**. It should click into place to ensure that it is the first thing that is done.



```
when space key pressed
delete 1 of itemsToSort
repeat 10
    add pick random 1 to 10 to itemsToSort
```

☑ Change the <1> to <All>. This will ensure that the list is cleared out before we try and add items to it. Doing this each time will mean the size of the list is always the same when demonstrating the program.



```
when space key pressed
delete 1 of itemsToSort
repea
    ad ndom 1 to 10 to itemsToSort
```

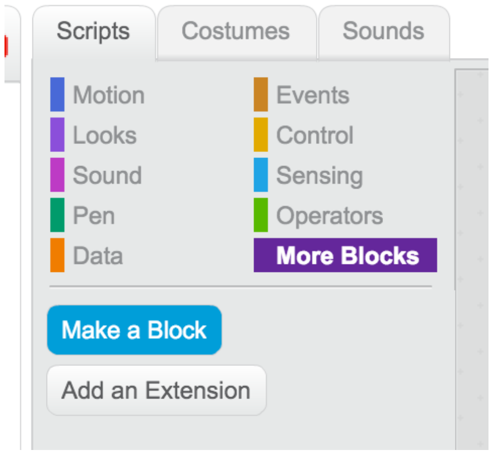
☑ Now retest the program by hitting the space bar. You should see that the list is cleared each time and rebuilt with our random numbers.

Step 4 - Create the Swap function

Now that we have a list to work with, we can start to build the algorithm. A good principle of coding is to build small pieces of code that can be tested and then combined to build the final program.

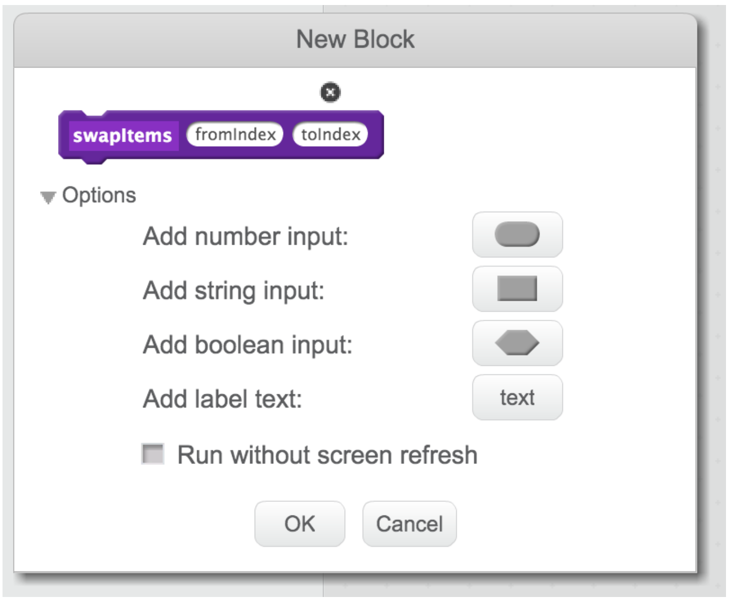
One of the basic processes required by our sorting algorithm is the ability to swap two items in the list. We will write a function to handle sorting, in Scratch functions are represented by Blocks. Let us now build the swap function.

☑ From the More Blocks section, click on **Make a Block**

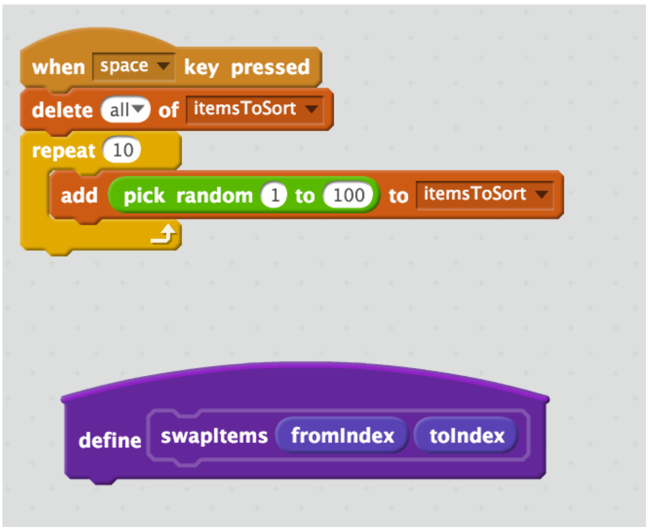


☑ Enter a name for the block. I have used **swapItems**. Expand the Options. In order to swap two items, we need to understand the algorithm for swapping two items in a list. The algorithm will operate on our **itemsToSort** list, it will require two parameters to be passed in

1. The From index - This is the position of one of the items being swapped
2. The To index - The is the position of the other item being swapped.



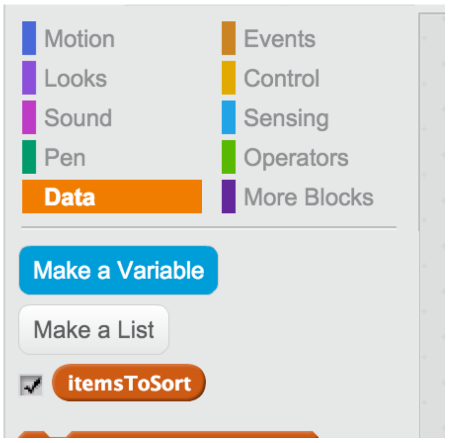
☑ A new puzzle piece will have appeared in the code area, it may be sat on top of your existing code, drag it to some free space so work can begin.



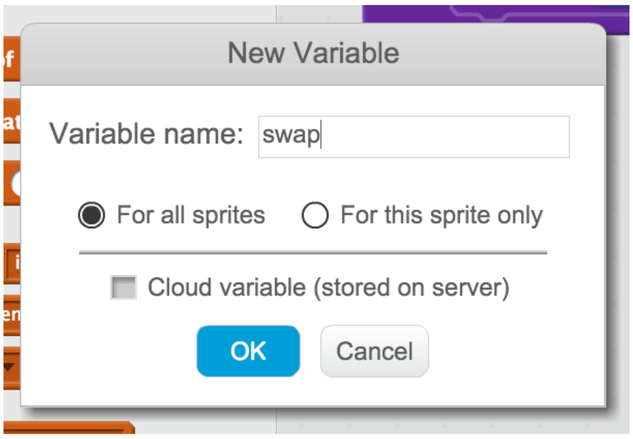
```
when [space] key pressed
delete (all) of [itemsToSort]
repeat (10)
    add (pick random (1) to (100)) to [itemsToSort]

define swapItems (fromIndex) (toIndex)
```

☑ Within the function, a third variable is required, which is used as temporary storage. Under Data, click on **Make a Variable**.



☑ Enter a name, I have used **swap**. Then click ok.



New Variable

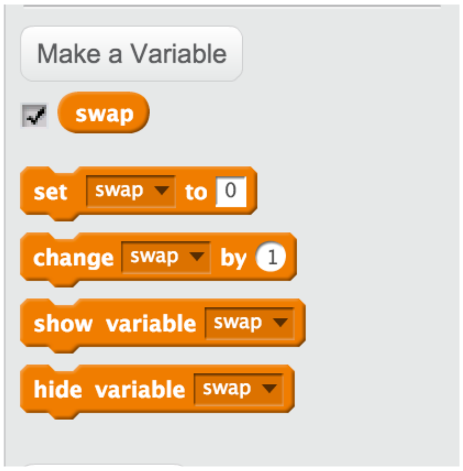
Variable name: swap

(● For all sprites) ○ For this sprite only

☐ Cloud variable (stored on server)

OK Cancel

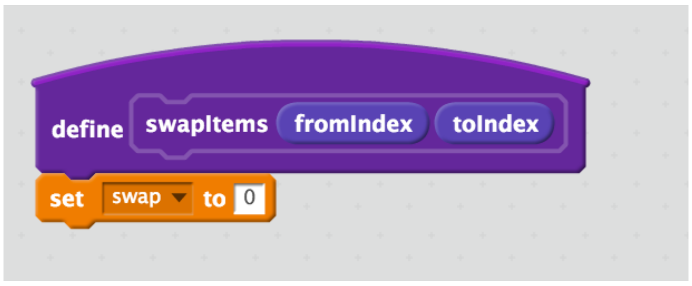
☑ Some new options will have appeared under the Data section, these are blocks that can be used to manipulate our variable.



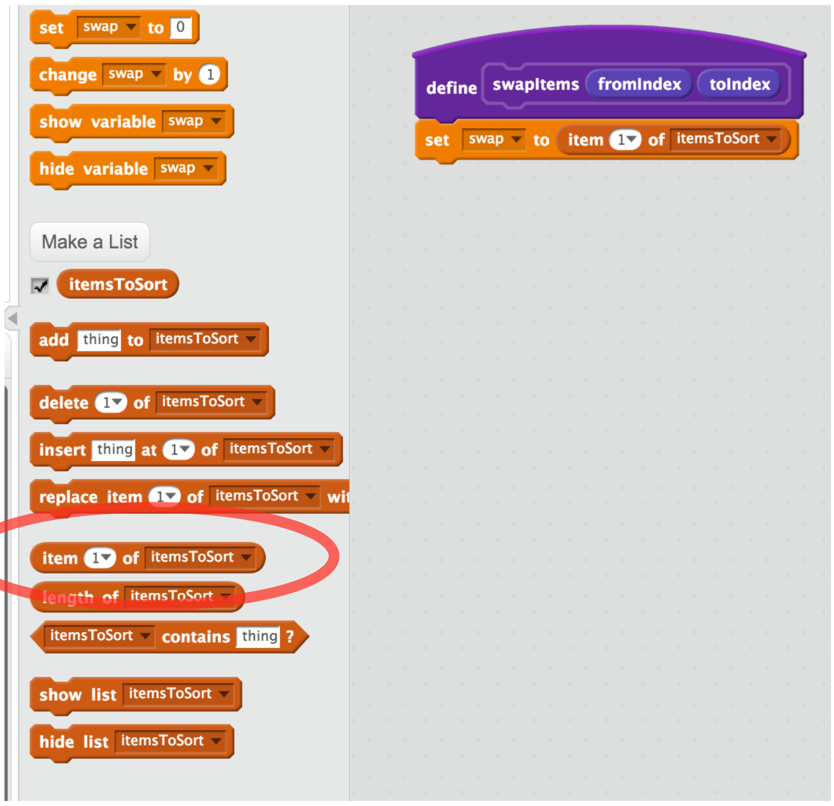
To understand why this swap variable is required, imagine holding a glass of water in each hand, and being told to swap the glasses between your hands. The easiest way to do this is to put one glass down on a table, pass the other glass to the empty hand, then pick up the original glass from the table. The table acts as the holding variable, it temporarily looks after one of the glasses of water whilst the other one is being passed between your hands. The algorithm will read as follows:

```
1. define swapItems (fromIndex, toIndex)
2.     swap := itemsToSort[fromIndex]
3.     itemsToSort[fromIndex] := itemsToSort[fromIndex + 1]
4.     itemsToSort[fromIndex+1] := swap
```

☑ Line one of the above is represented by our new swapItems block with two arguments. To define line 2, we must set the **swap** variable to the value of the item at position **fromIndex** in the **itemsToSort** list. From the Data section, drag the **Set <swap> to 0** block under our **swapItems** block.

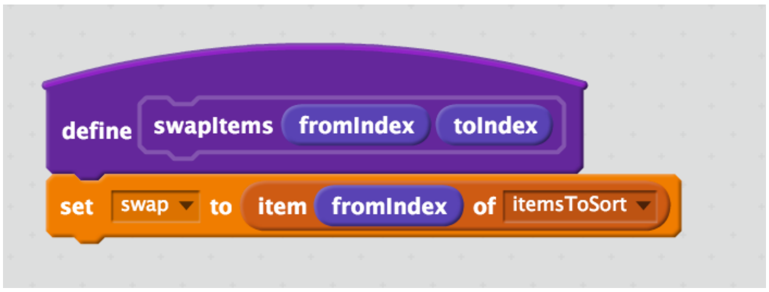


☑ Instead of setting swap to 0, we must set it to the item in the list at fromIndex. In order to get that value, drag **item <1> of <itemToSort>** block from the Data menu, to swap value.



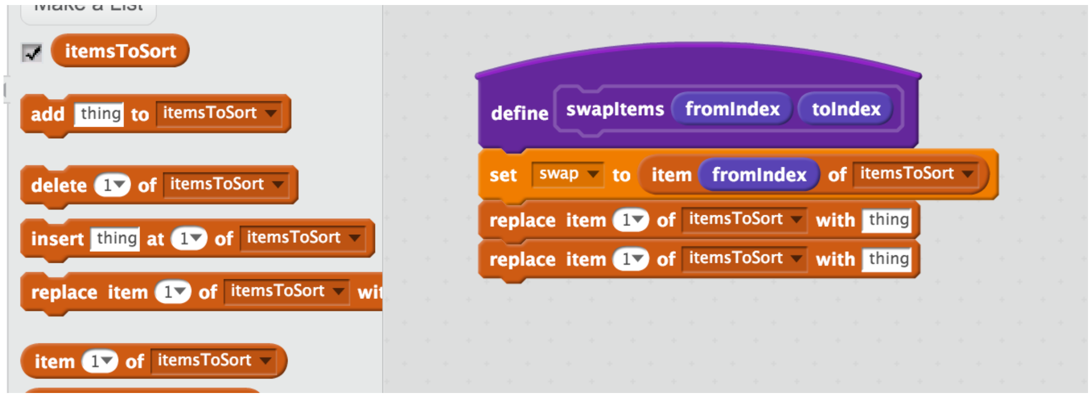
```
define swapItems fromIndex toIndex
set swap to item 1 of itemsToSort
```

☑ This will set swap to the first item in the list, we now need to drag the **fromIndex** variable into the item drop down.



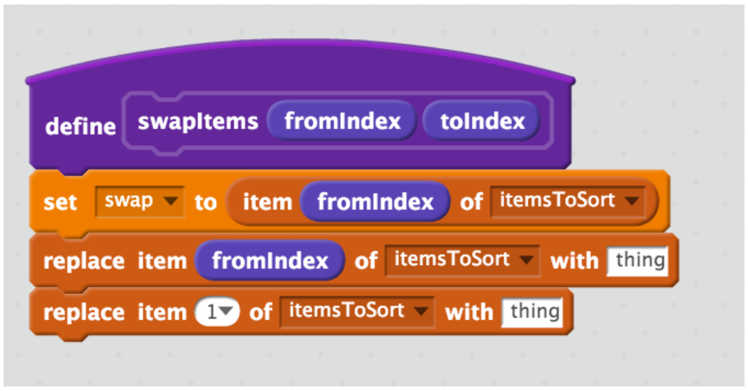
```
define swapItems fromIndex toIndex
set swap to item fromIndex of itemsToSort
```

☑ For lines 3 and 4, drag the **replace item <1> of <itemsToSort> with <thing>** to the bottom of our function, do this twice.



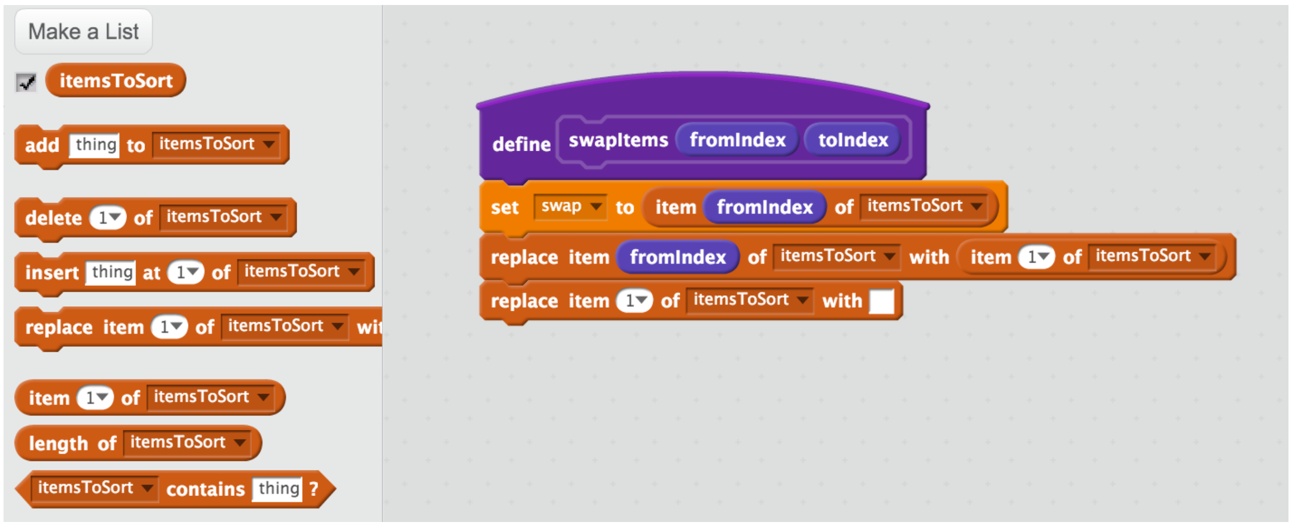
```
define swapItems fromIndex toIndex
set swap to item fromIndex of itemsToSort
replace item 1 of itemsToSort with thing
replace item 1 of itemsToSort with thing
```

☑ We need to drag the **toIndex** into the first replace block, and the **fromIndex** into the second replace block.



```
define swapItems fromIndex toIndex
set swap to item fromIndex of itemsToSort
replace item fromIndex of itemsToSort with thing
replace item 1 of itemsToSort with thing
```

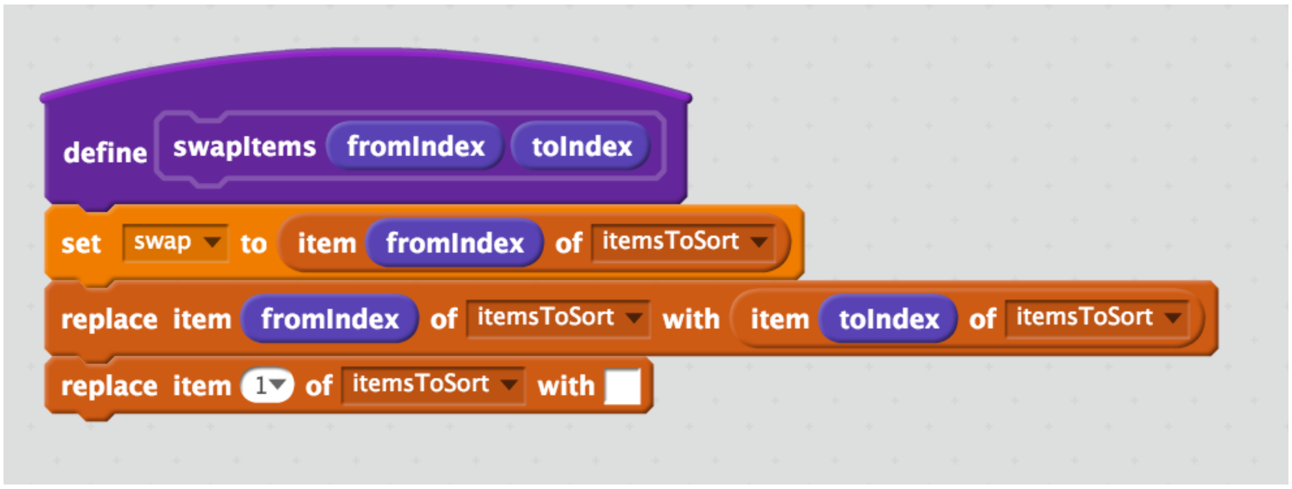
☑ The first replace block will take the value in the **toIndex** position, and place it in the **fromIndex** position. So drag the **item <1> of <itemsToSort>** block into the **thing** space of the first replace block.



```
Make a List
itemsToSort
add thing to itemsToSort
delete 1 of itemsToSort
insert thing at 1 of itemsToSort
replace item 1 of itemsToSort wi
item 1 of itemsToSort
length of itemsToSort
itemsToSort contains thing ?
```

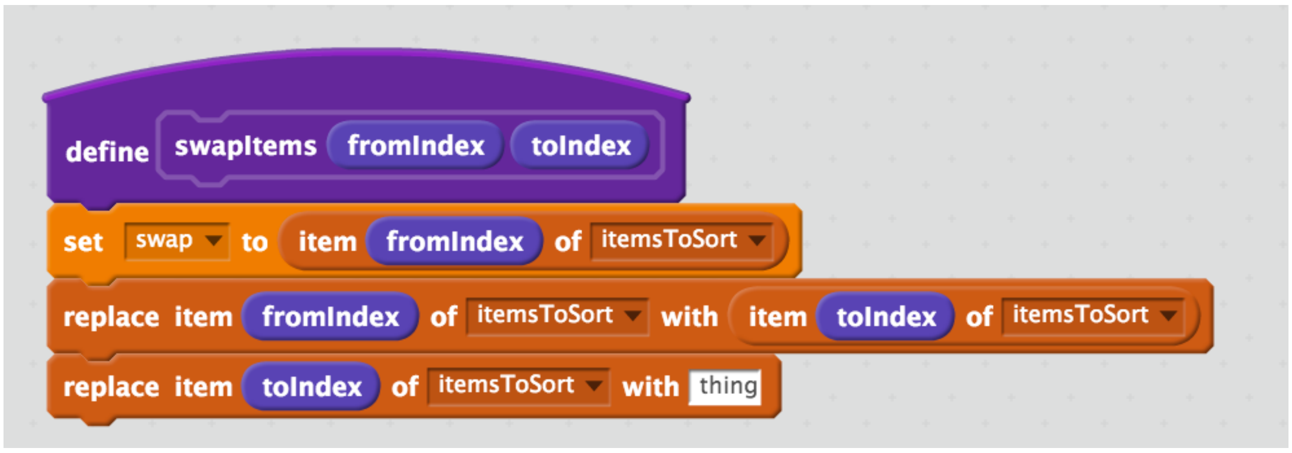
```
define swapItems fromIndex toIndex
set swap to item fromIndex of itemsToSort
replace item fromIndex of itemsToSort with item 1 of itemsToSort
replace item 1 of itemsToSort with
```

☑ Then drag the **toIndex** variable into the **1** space of the **item <1> of itemsToSort** block.



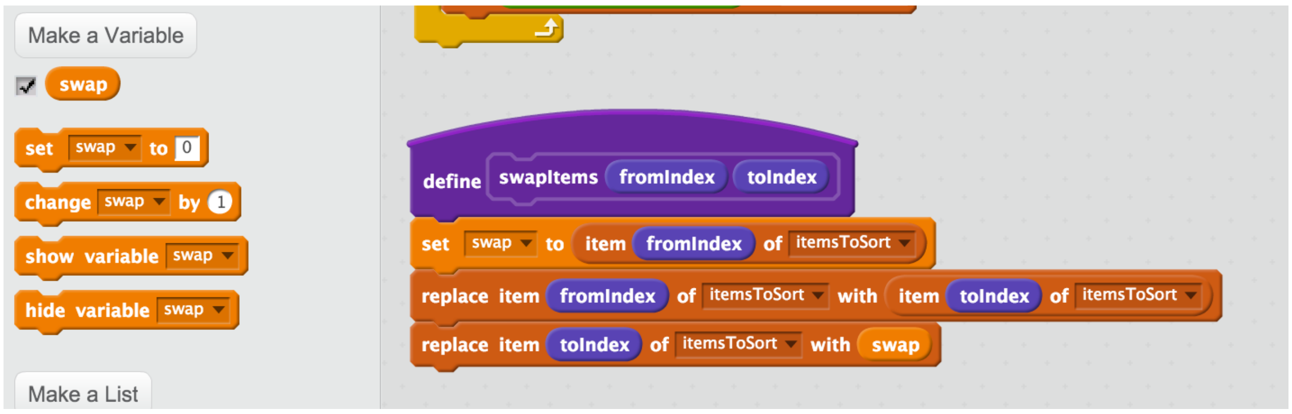
```
define swapItems fromIndex toIndex
set swap to item fromIndex of itemsToSort
replace item fromIndex of itemsToSort with item toIndex of itemsToSort
replace item 1 of itemsToSort with
```

☑ Now for the second replace block, the item we are replacing is the one in **toIndex** so drag **toIndex** into the **replace item <1>** space.



```
define swapItems fromIndex toIndex
set swap to item fromIndex of itemsToSort
replace item fromIndex of itemsToSort with item toIndex of itemsToSort
replace item toIndex of itemsToSort with thing
```

☑ This is where the swap value comes in, we now drag the **swap** variable from the Data section, into the **with <thing>** space in the second replace block.



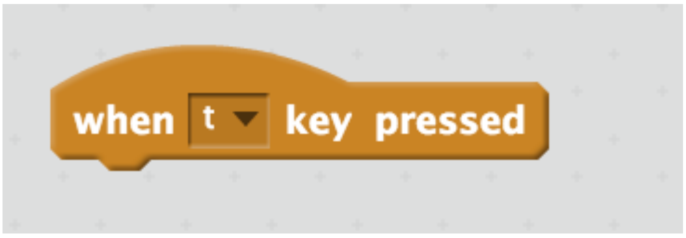
```
define swapItems fromIndex toIndex
set swap to item fromIndex of itemsToSort
replace item fromIndex of itemsToSort with item toIndex of itemsToSort
replace item toIndex of itemsToSort with swap
```

The swapping function is now complete, so the next section will show you how to test it.

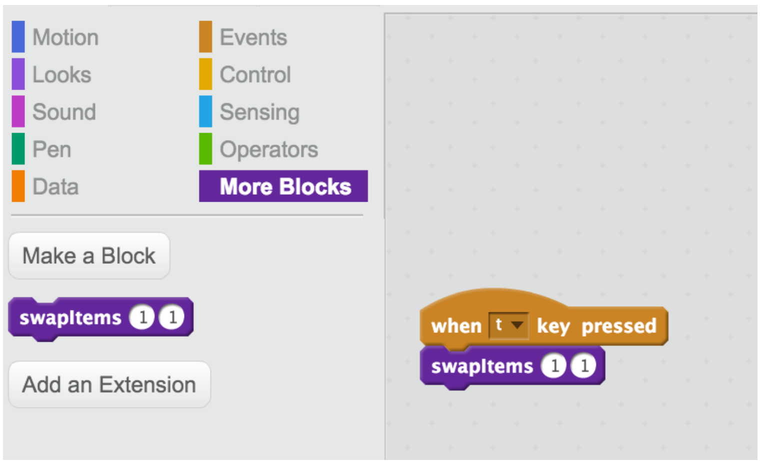
Step 5 - Test the Swap Function

In order to test the function, we must call it from another code block. We will create a new code block that runs when the user presses <t> (for test). This test function will call our swap function with some fixed values and we can observe the list changing. This will allow us to see that our swap function is working properly.

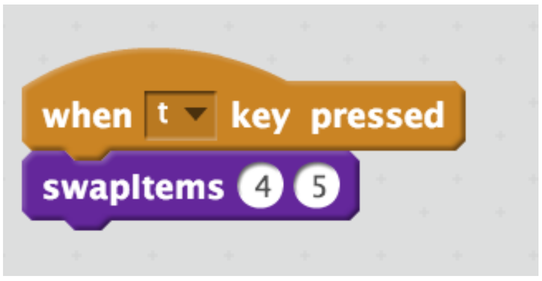
☑ From Events, drag a **When <space> key pressed** block into the code area. Change the key to <t> (instead of <space>)



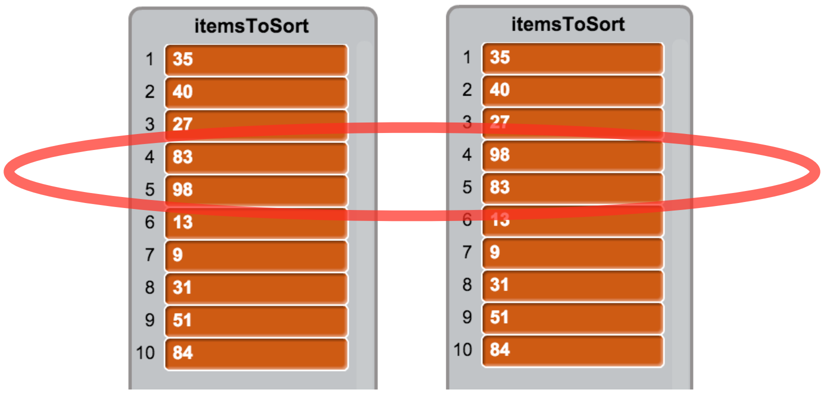
☑ From More Blocks, drag our **swapItems** block underneath the key press.



☑ Set the two numbers to something other than 1 and 1. The screenshot shows us attempting to swap items 4 and 5.



☑ Now try pressing the <t> key and observe the list in the game display. If you keep pressing <t> then the 2 items at the indexes should swap each time. If they do, then congratulations! Your swap function works. Feel free to experiment with the values passed to swapItems.



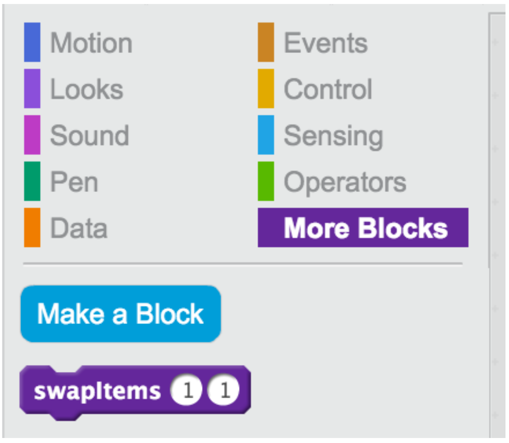
	itemsToSort
1	35
2	40
3	27
4	83
5	98
6	13
7	9
8	31
9	51
10	84

	itemsToSort
1	35
2	40
3	27
4	98
5	83
6	13
7	9
8	31
9	51
10	84

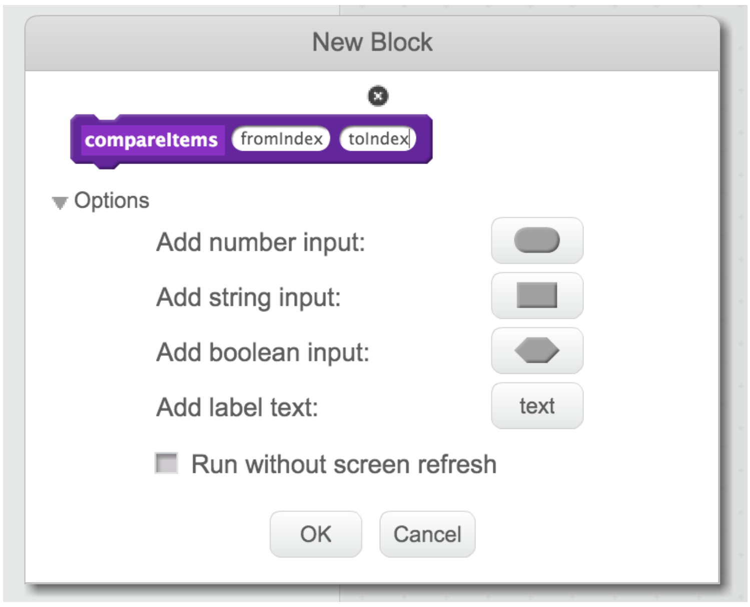
Step 6 - The Compare Function

For the sorting algorithm to work, it should only swap items that are in the wrong order. In this section we will write a new function which compares two numbers in our list, and determines if they are in the correct order. If they are not in the correct order, then they will be swapped.

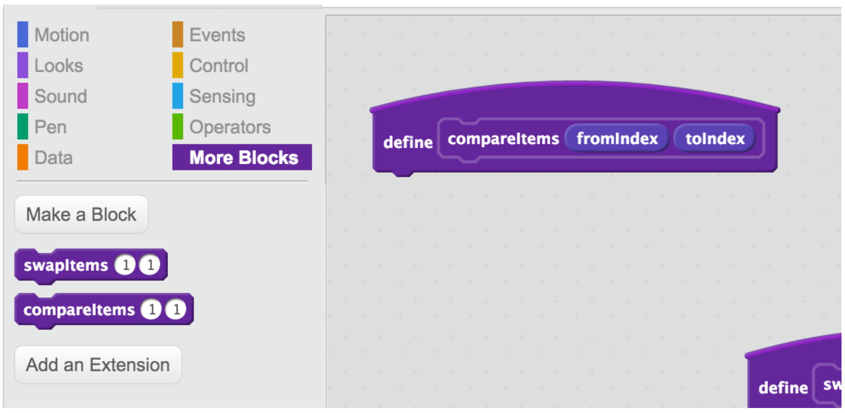
☑ In More Blocks, click on Make a Block.



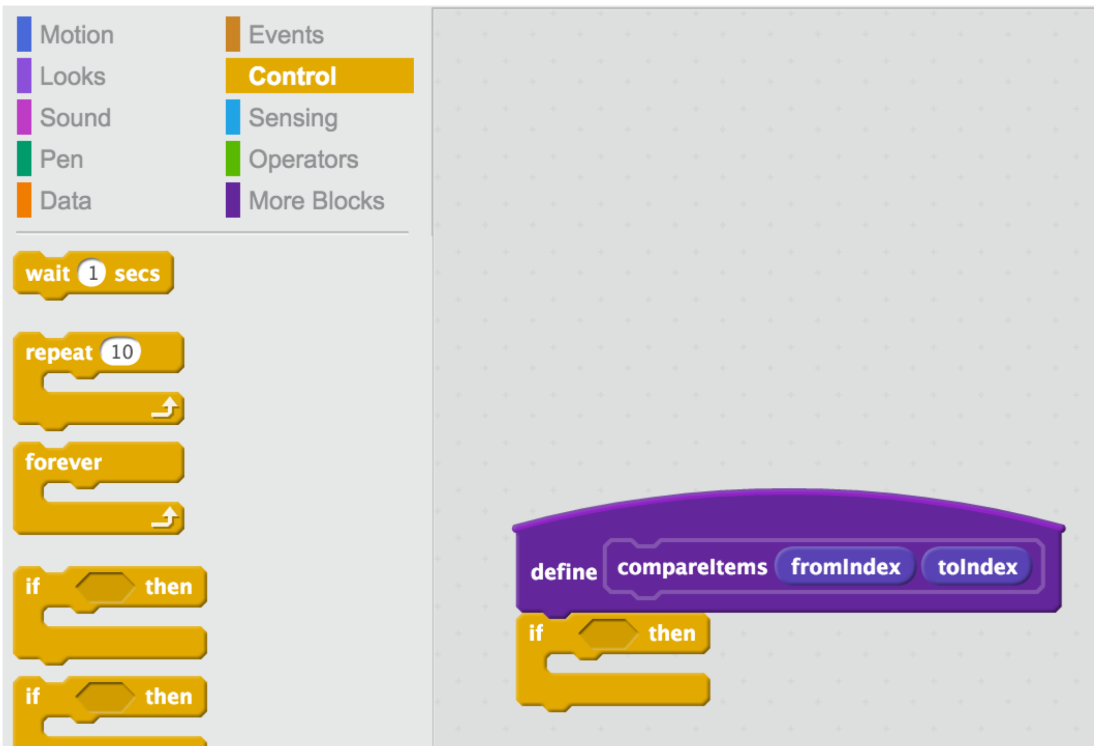
☑ Type the name of the new block, I have used **compareItems**. This block will require two parameters, so click on **Add Number Input** twice, then give each of these number inputs a name. I have used **fromIndex** and **toIndex** which is the same as my swap function.



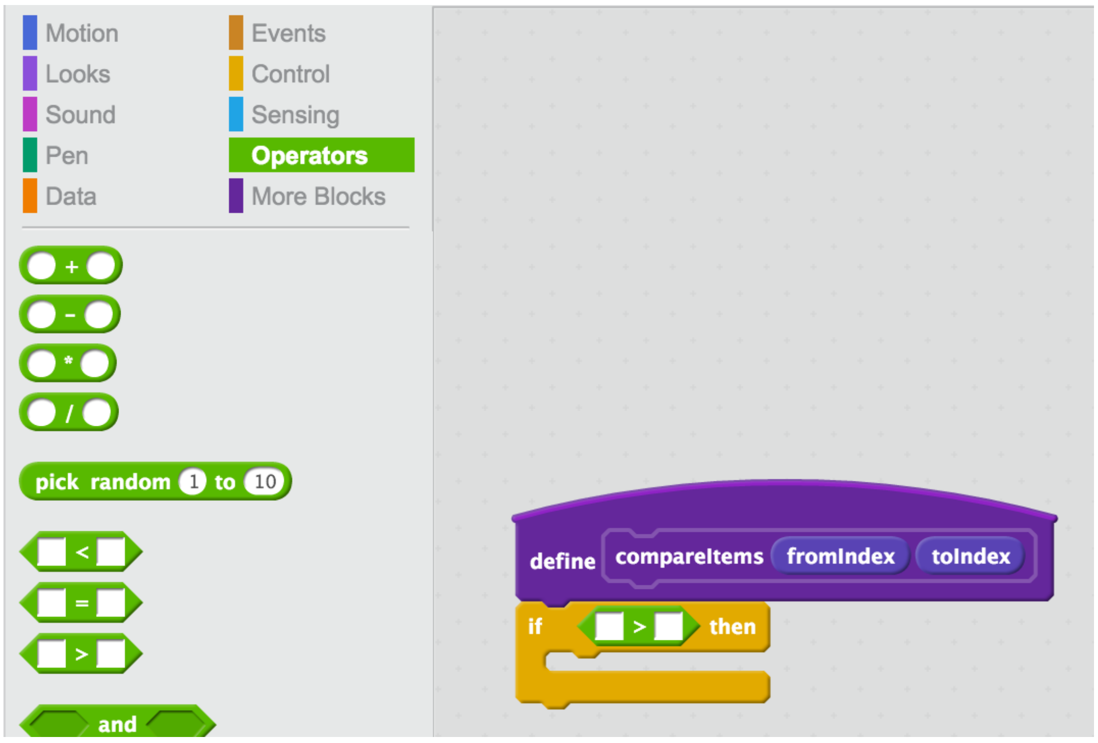
☑ The new block should appear in the code area, with a new piece that can be dragged on under More Blocks.



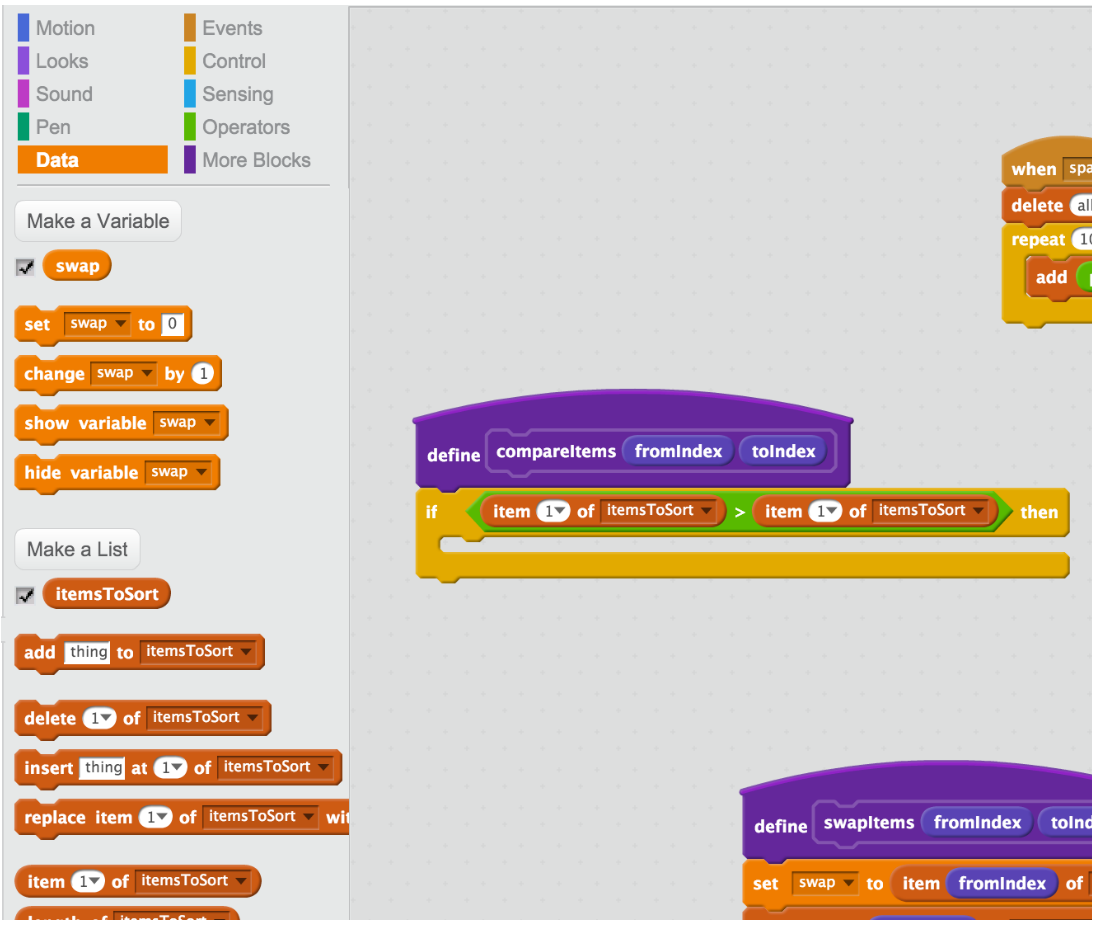
☑ The comparison will require an **if statement**. From Control, drag on of these blocks into our new function.



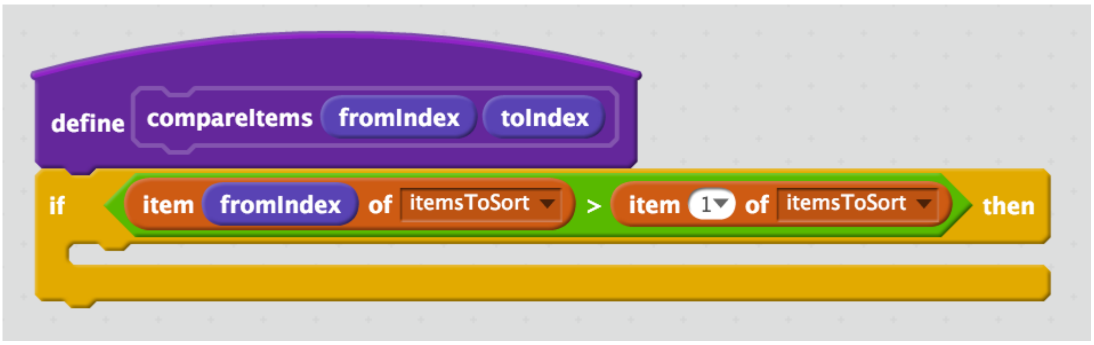
☑ Inside the **if statement**, a comparison will be required. The comparison will check to see if to **fromIndex** item in our list is bigger than than the **toIndex** item. If it is then the swap will be called. From Operators, drag the **more than** block into our **if statement**.



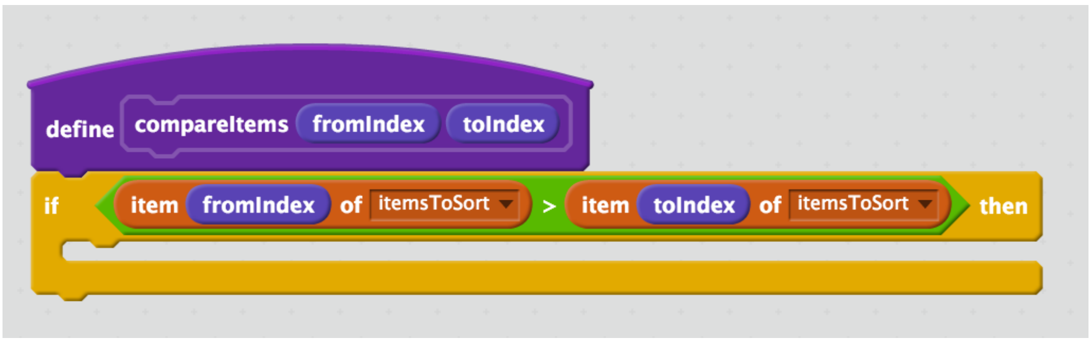
☑ Both of the arguments to this operator will be items on our list. So from Data drag **item <1> of <itemsToSort>** into each of the spaces in our **more than** block.



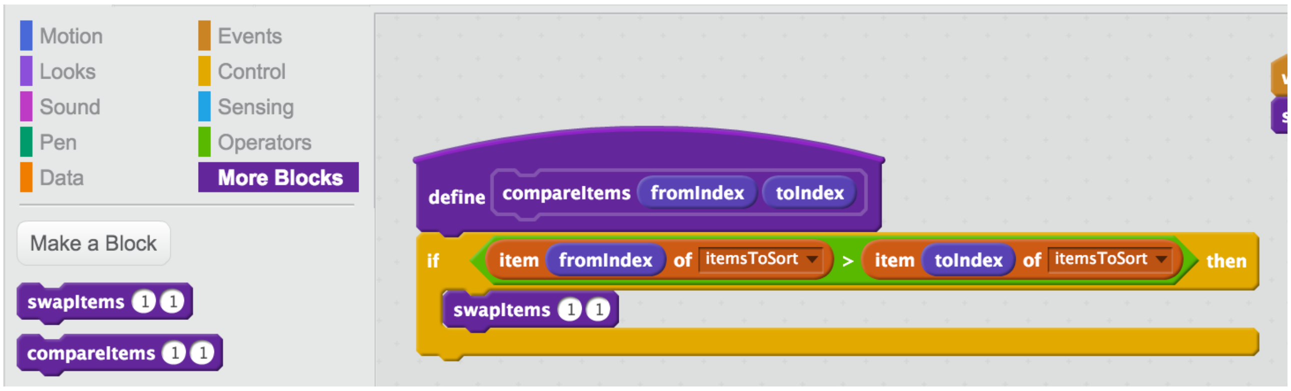
☑ The indexes of these items needs to be set, the indexes are those that are passed into the **compareItems** function. First drag the **fromIndex** into the first of our **item <1> of <itemsToSort>** blocks.


define compareItems fromIndex toIndex
if item fromIndex of itemsToSort > item 1 of itemsToSort then

☑ Then drag **toIndex** into the second of our **item <1> of <itemsToSort>** blocks.

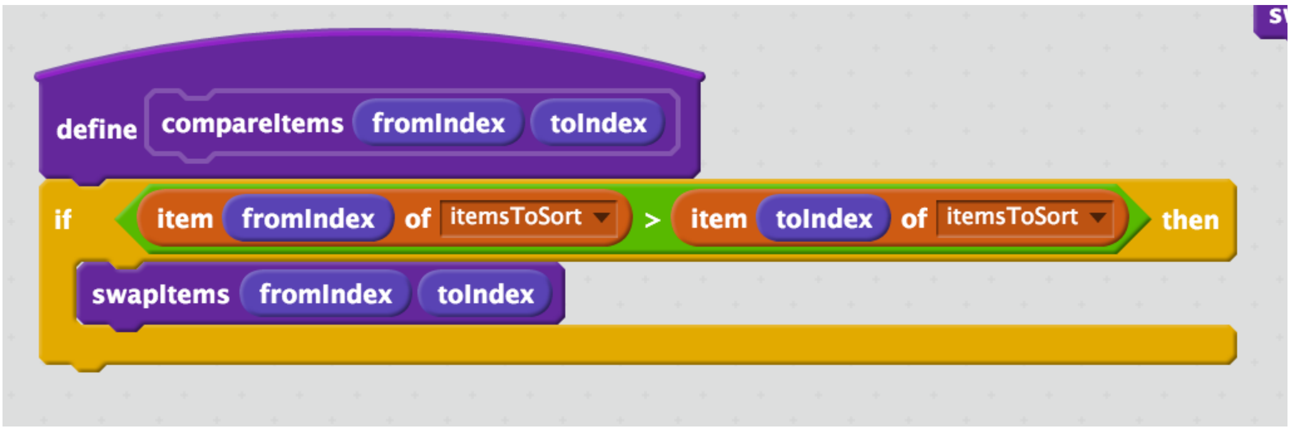

define compareItems fromIndex toIndex
if item fromIndex of itemsToSort > item toIndex of itemsToSort then

☑ If the condition passes, then we must swap the items. So from More Blocks, drag our **swapItems** function inside the **if statement**.



```
define compareItems fromIndex toIndex
if <(item fromIndex of itemsToSort) > (item toIndex of itemsToSort)> then
    swapItems 1 1
```

☑ To swap the correct items, we must drag **fromIndex** and **toIndex** into the arguments for **swapItems**.

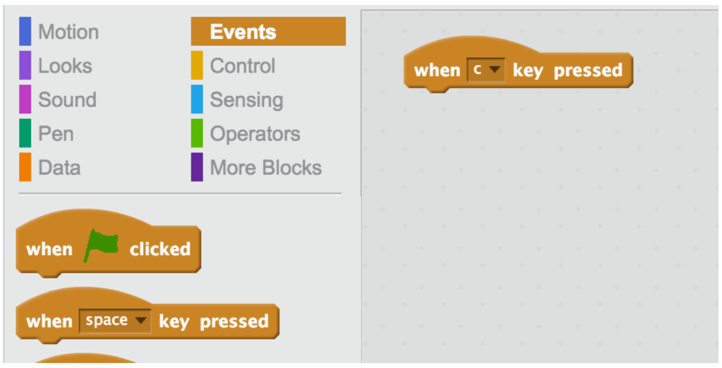


```
define compareItems fromIndex toIndex
if <(item fromIndex of itemsToSort) > (item toIndex of itemsToSort)> then
    swapItems fromIndex toIndex
```

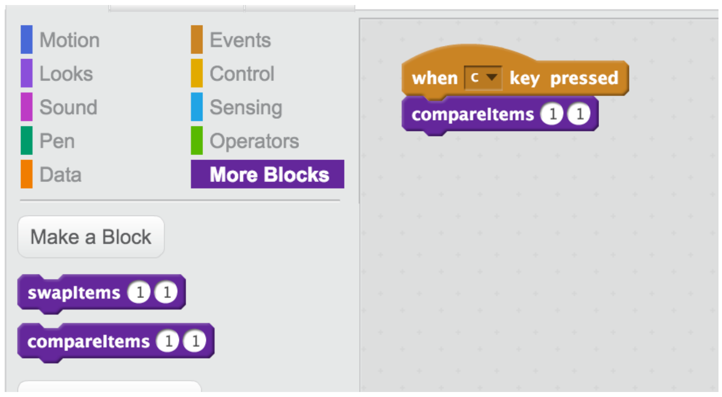
Step 7 - Test the Compare Function

Now that a compare function exists, we should test it. This is the same structure used for Step 5 where we tested the Swap function. We are going to call the Compare function with fixed numbers for **fromIndex** and **toIndex** and see if the program behaves correctly.

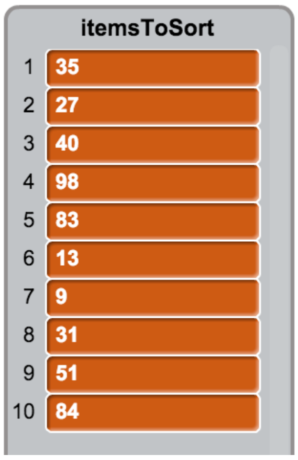
☑ From Events, drag a **When <space> key pressed** block into the code area. Change the key to <c> (instead of <space>)



☑ from More Blocks, drag our **compareItems** block into this new function.

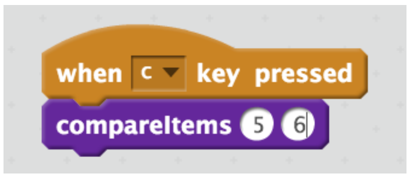


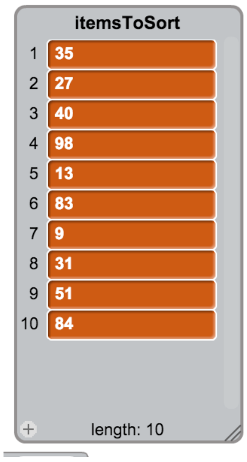
☑ Pick two numbers in the list that are in the wrong order, then put those indexes into your function. Try hitting the <c> key and the function should swap the items. In the example below, items 5 and 6 start off in the wrong order. When I hit <c> the **compareItems** can see they are in the wrong order and calls the swap function. Remember that you can always generate a new random list using the <space> bar if you can't find any examples in your current data.



itemsToSort

1	35
2	27
3	40
4	98
5	83
6	13
7	9
8	31
9	51
10	84





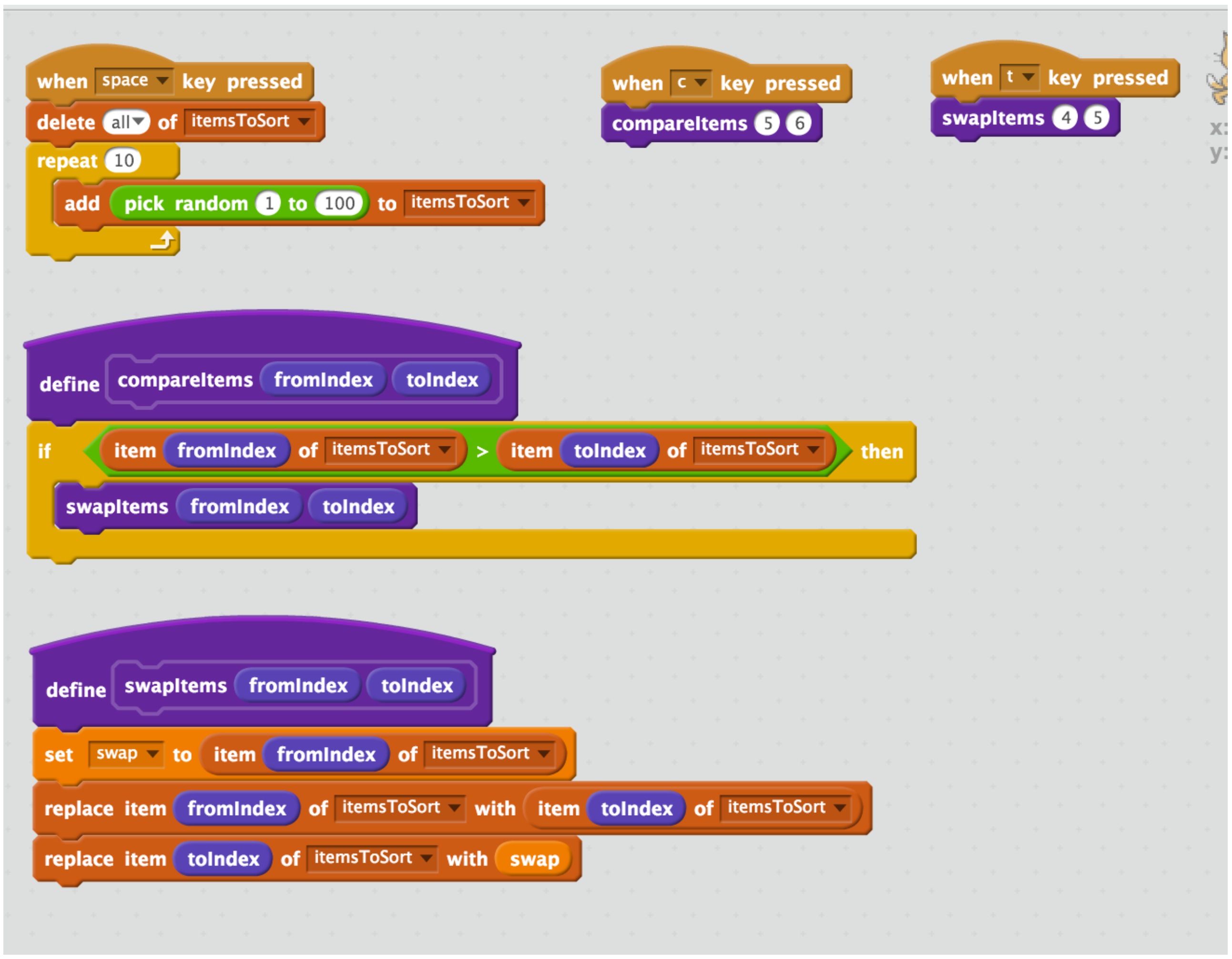
itemsToSort

1	35
2	27
3	40
4	98
5	13
6	83
7	9
8	31
9	51
10	84

length: 10

Step 8 - Recap

Before we move on, let’s have a look at our entire code base.



```
when space key pressed
delete all of itemsToSort
repeat 10
    add pick random 1 to 100 to itemsToSort

when c key pressed
compareItems 5 6

when t key pressed
swapItems 4 5

define compareItems fromIndex toIndex
if item fromIndex of itemsToSort > item toIndex of itemsToSort then
    swapItems fromIndex toIndex

define swapItems fromIndex toIndex
set swap to item fromIndex of itemsToSort
replace item fromIndex of itemsToSort with item toIndex of itemsToSort
replace item toIndex of itemsToSort with swap
```

We should have the following 5 things

1. Function to generate a new list when hitting <space>
2. Function to compare items
3. Function to swap items
4. Test for the compare items function
5. Test for the swap items function

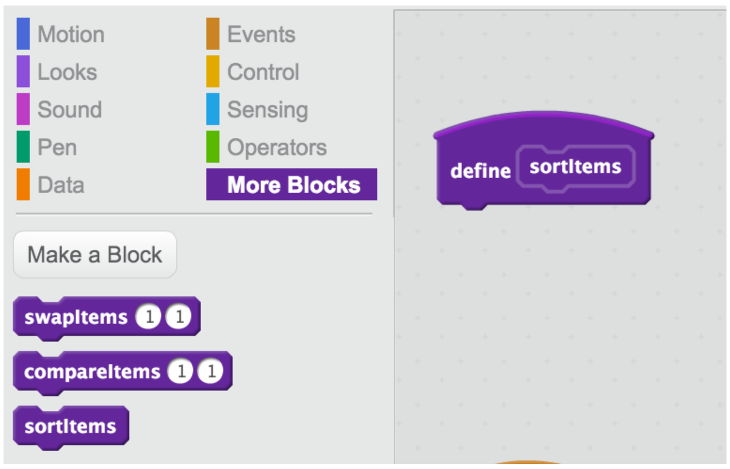
If all of these things are correct, we can move onto the next step. We have one more function to write and one more test!

Step 9 - Write the Iteration Function

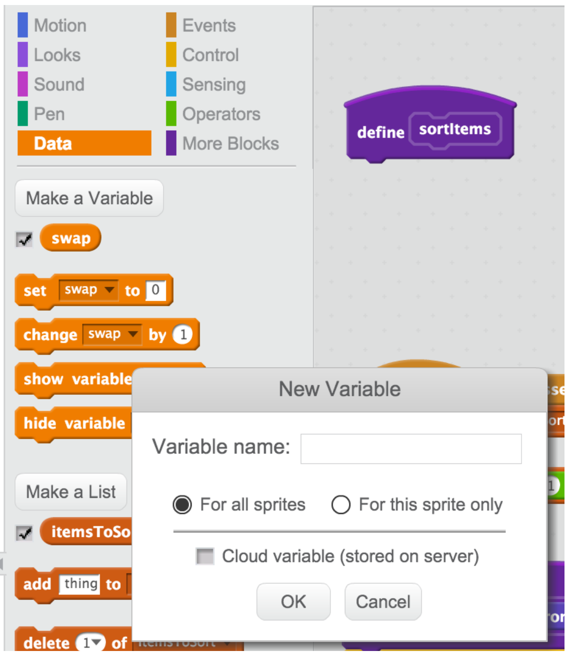
The Bubble Sort algorithm uses a particular set of comparisons and swaps in order to sort the entire list. It must repeat the iteration process until the list is sorted. On each loop through, the highest number will end up in its correct place (it bubbles upwards). So on each loop through we go through 1 less item until we no longer need to run any comparisons.

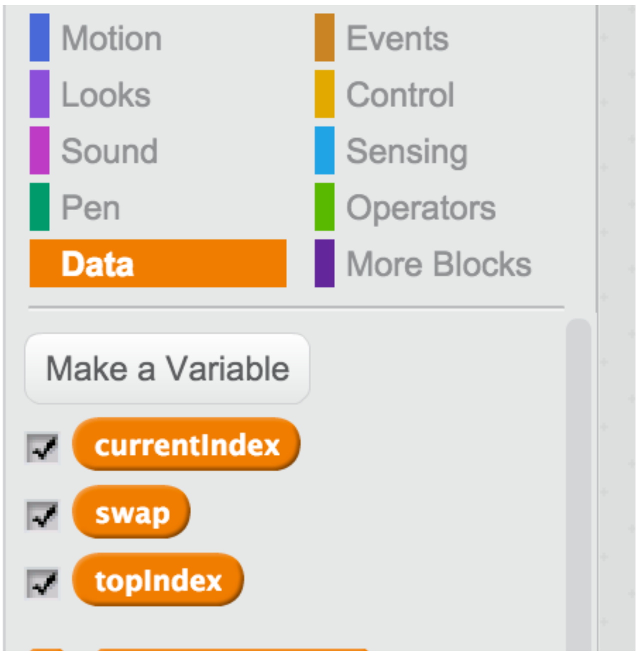
This algorithm requires an outer loop, which shortens the run through each time, and an inner loop, which counts up through the list, sorting as it goes along.

☑ Create a new block for our sort function, from More Blocks, click on **Make a Block**, type a name **sortItems** and click OK. This block does not require any arguments.

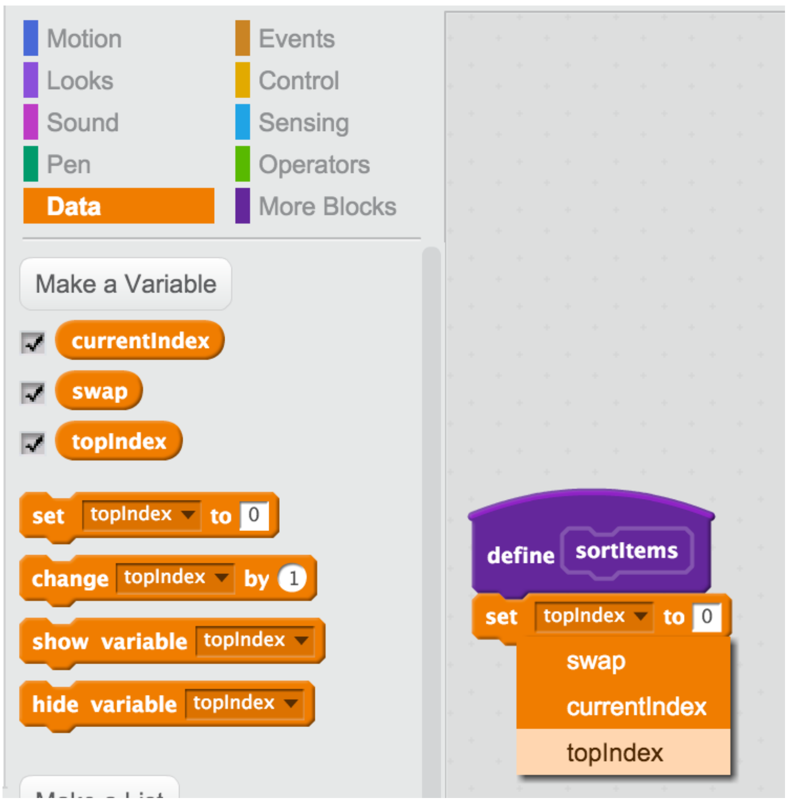


☑ We need two new variables, one to track the **currentIndex** of our current iteration, and one to track the **topIndex** to track how far up the list we still have to go. Under Data click on **Make a Variable**, give the name (currentIndex for the first one, topIndex for the second one)

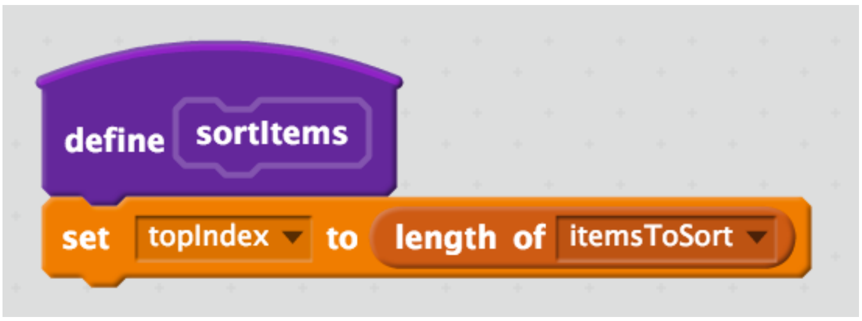




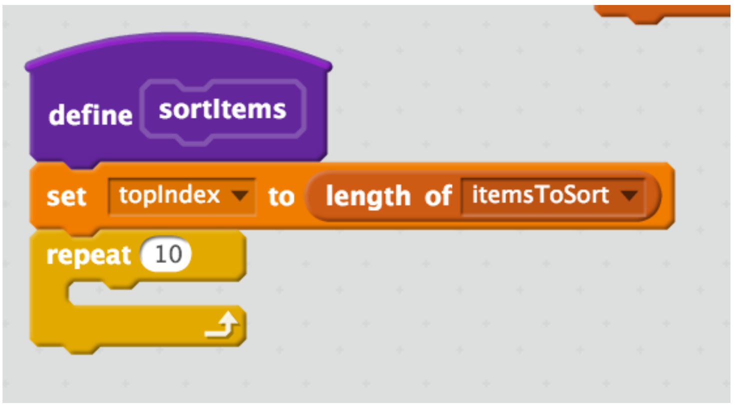
☑ At the start of the algorithm, the **topIndex** must be set equal to the number of items in our list. From Data, drag **set <topIndex> to <0>** into our new function. Ensure that the variable being set is **topIndex** by using the drop down.



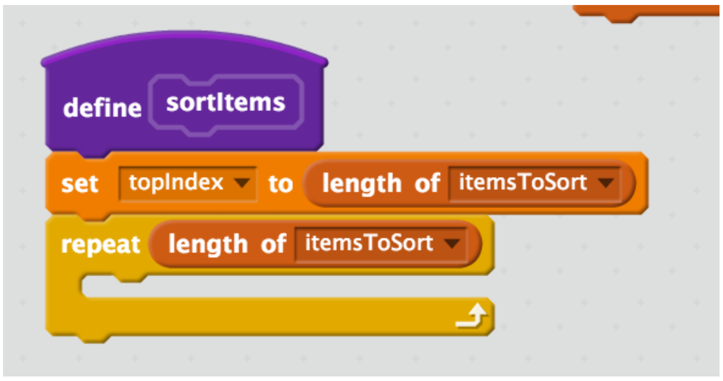
☑ Set the topIndex to **length of <itemsToSort>** by dragging that block from Data into the set variable.



☑ Now for the outer loop, from Control, drag a **repeat <10>** block into our function.

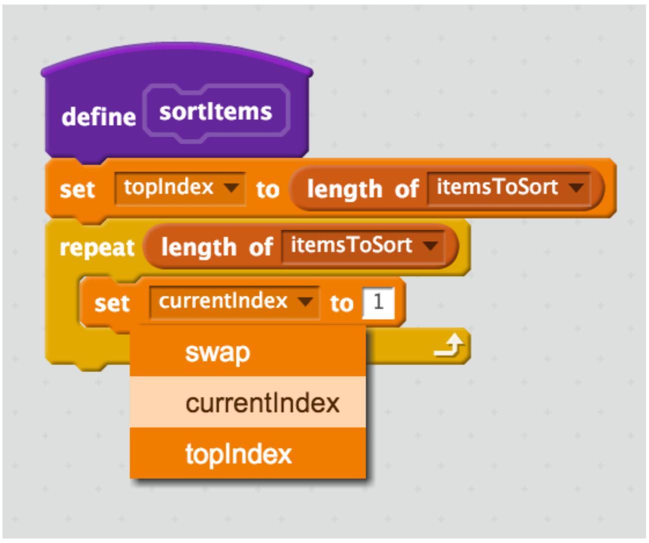


☑ The repeat should run through for the number of items in our list. From Data drag **length of <itemsToSort>** into the range variable for our **repeat <10>** block.



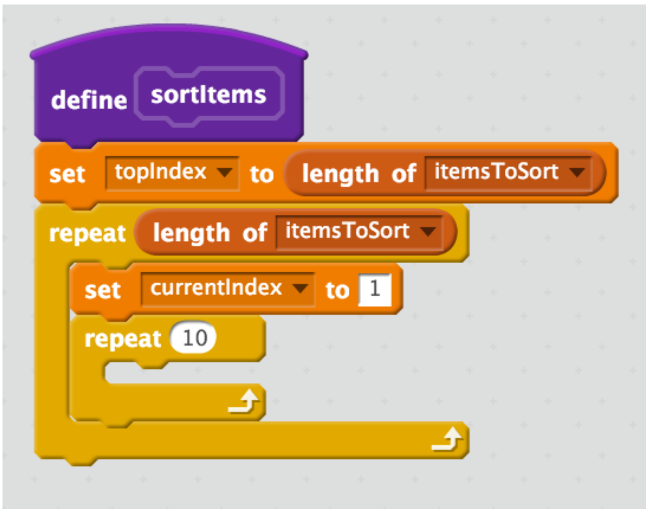
```
define sortItems
set topIndex to (length of itemsToSort)
repeat (length of itemsToSort)
end
```

☑ First thing inside the loop is to set the **currentIndex** to zero, this points the **currentIndex** to the first item in our list. From Data, drag **set <currentIndex> to <1>** into our loop. Ensure that it is **currentIndex** by using the drop down. Set the value to **1**, this ensures the index points to the first item in our list.



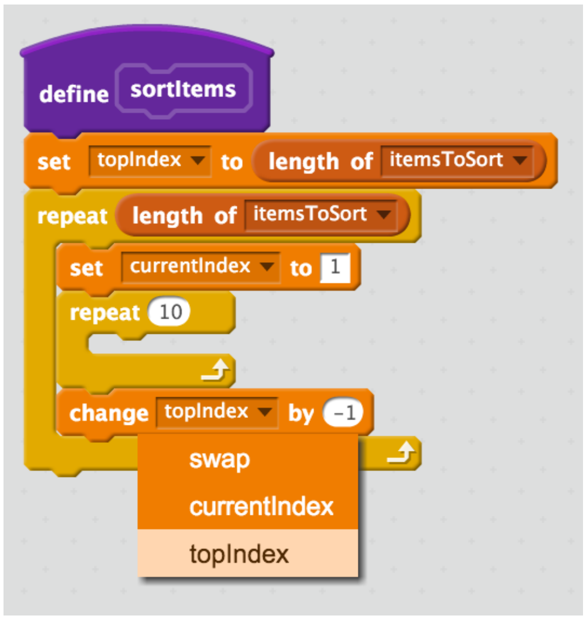
```
define sortItems
set topIndex to (length of itemsToSort)
repeat (length of itemsToSort)
    set currentIndex to 1
end
```

☑ Create the inner loop by dragging in another **repeat <10>** block just after setting **currentIndex** to zero.



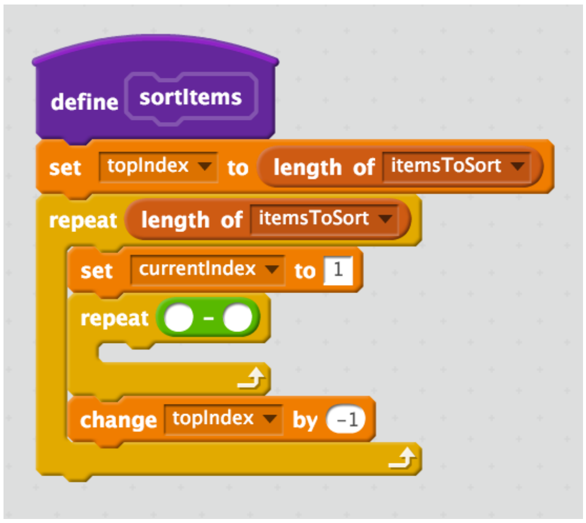
```
define sortItems
set topIndex to (length of itemsToSort)
repeat (length of itemsToSort)
    set currentIndex to 1
    repeat 10
    end
end
```

☑ After the inner loop, the **topIndex** should be reduced by 1 so that the outer loop is counting backwards. From Data, drag **change <topIndex> by <-1>** block just after our inner loop. Make sure to set the variable to **topIndex** using the drop down, and set the **changeBy** to **-1** instead of **1**.



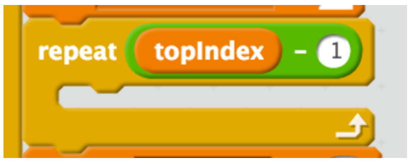
```
define sortItems
set topIndex to length of itemsToSort
repeat length of itemsToSort
    set currentIndex to 1
    repeat 10
    
    change topIndex by -1
```

☑ Now we can work on the inner loop. The limit of the inner loop should be **topIndex -**1. The reason we should subtract 1, is because the comparison will compare the current item and the one after it. For this bit of maths, from Operators drag the subtract block into our range.

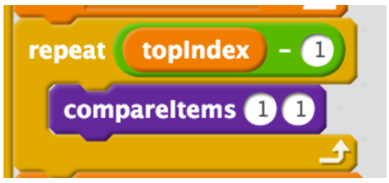


```
define sortItems
set topIndex to length of itemsToSort
repeat length of itemsToSort
    set currentIndex to 1
    repeat ( ) - ( )
    
    change topIndex by -1
```

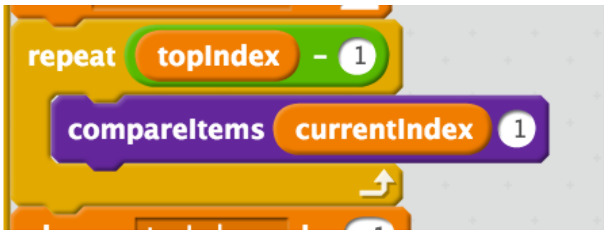
- The first argument for the subtraction is **topIndex**, the second argument is **1**. Drag **topIndex** into our subtraction and type in 1.



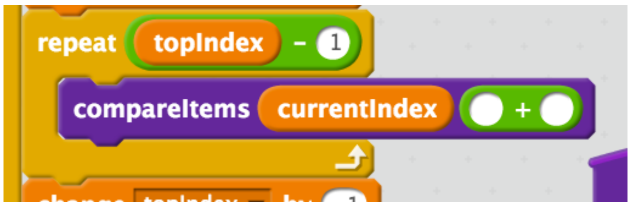
- Inside this inner loop, we now call our **compareItems** function. From More Blocks, drag in **compareItems**.



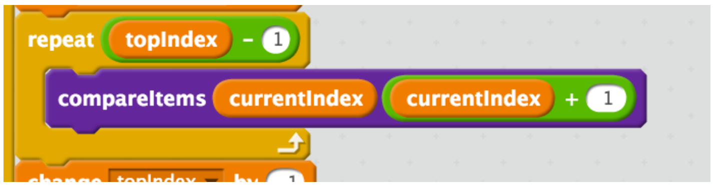
- Set the first argument to be **currentIndex** by dragging it from Data.



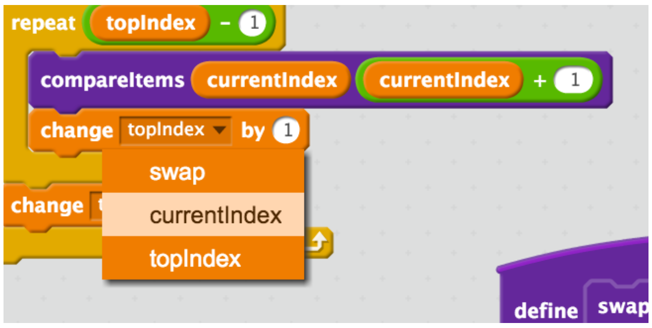
- The second argument will be **currentIndex + 1** so drag the addition operator into the second argument.



- Drag **currentIndex** into the addition and type in 1.



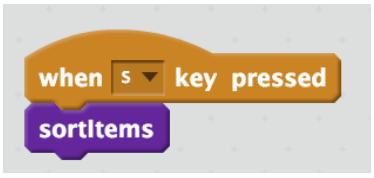
- After running the comparison, the **currentIndex** should be incremented (+1) so that the loop can count upwards. From Data drag **change <currentIndex> by 1** to just after our comparison. This completes the sorting function.



Step 10 - Test the Sorting Function

The last step is to simply call the sorting function when the player presses a key. This is the same procedure as the other test functions.

- ☑ From Events, drag **When <space> is pressed** to the code area, change the key to **<s>** (for sort).
- ☑ From More Blocks, drag our **sortItems** function into this event handler.

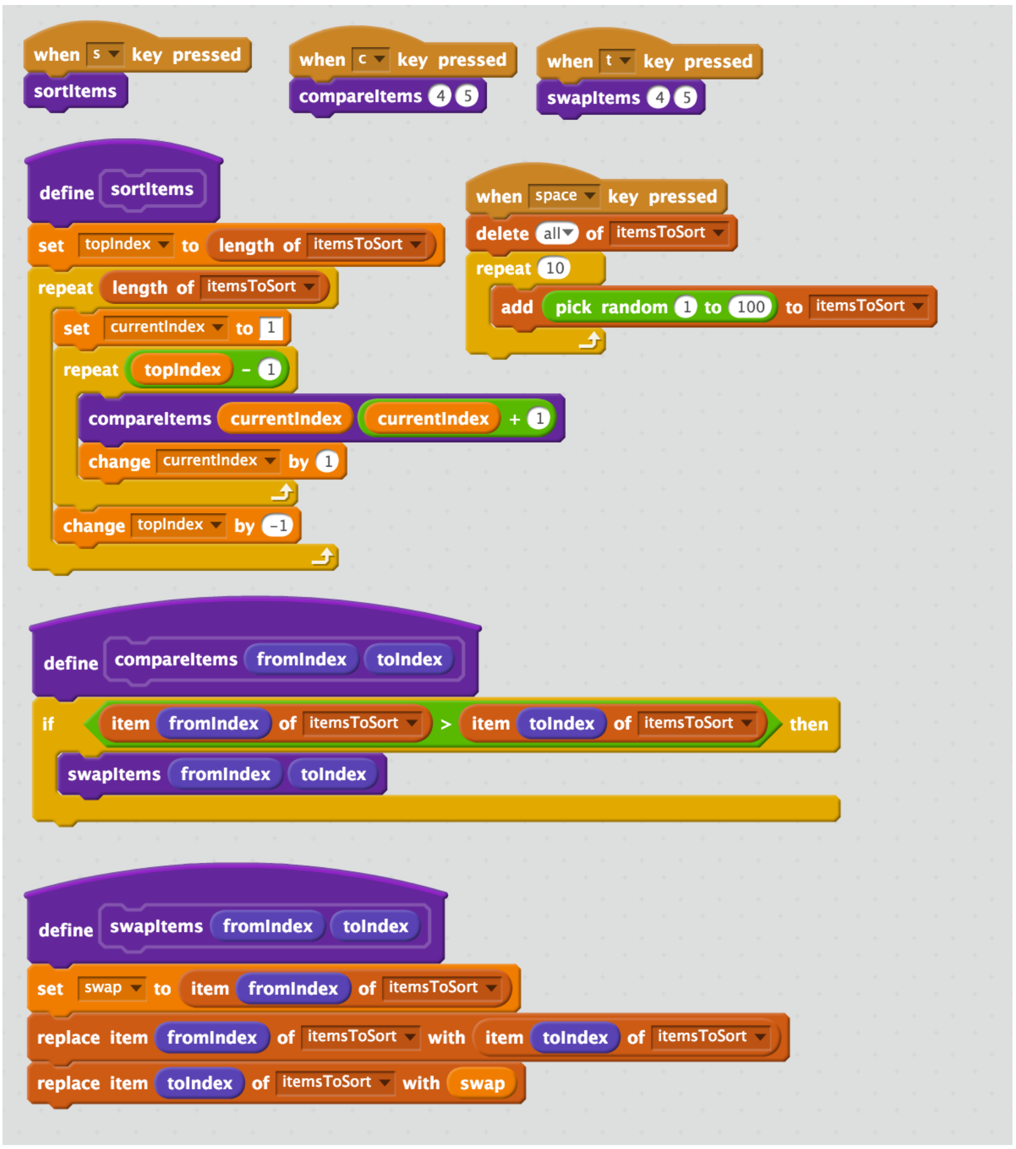


```
when s key pressed
sortItems
```

The code should now look like the following. We have 7 functions.

1. Function to generate list on pressing <space>
2. Function to test swap items on pressing <t>
3. Function to test comparison on pressing <c>
4. Function to test sorting algorithm on pressing <s>
5. Sorting Function
6. Comparison Function
7. Swapping Function

Congratulations, you have now implemented the Bubble Sorting algorithm in Scratch!



```
when s key pressed
sortItems

when c key pressed
compareItems 4 5

when t key pressed
swapItems 4 5

define sortItems
set topIndex to length of itemsToSort
repeat length of itemsToSort
    set currentIndex to 1
    repeat topIndex - 1
        compareItems currentIndex (currentIndex + 1)
        change currentIndex by 1
    change topIndex by -1

when space key pressed
delete all of itemsToSort
repeat 10
    add pick random 1 to 100 to itemsToSort

define compareItems fromIndex toIndex
if item fromIndex of itemsToSort > item toIndex of itemsToSort then
    swapItems fromIndex toIndex

define swapItems fromIndex toIndex
set swap to item fromIndex of itemsToSort
replace item fromIndex of itemsToSort with item toIndex of itemsToSort
replace item toIndex of itemsToSort with swap
```