Snap! Activity - Fundamental Coding Concepts

CS4S Maths - Networks Workshop

# Introduction

The fundamental *Coding* concepts we will introduce you to in this activity are:

* *Algorithms and Programs*
* *Sequencing*
* *Repetition*
* *Variables*
* *Functions*
* *User Input*
* *Branching*
* *Lists*

# The Project

## Turtle Geometry

The design of this tutorial has been inspired by *Turtle Geometry*, a book written by Hal Abelson and Andrea DiSessa in the 1980s. *Turtle Geometry* is a book about learning university-level mathematics with the *LOGO* programming language, which was one of the first programming languages designed for educational purposes. *Snap!* has blocks that allow you to move a "Turtle" around the canvas and draw shapes with a pen tool, just like you could in *LOGO*.

We will create a *Snap!* project that combines all the concepts mentioned in the previous section with Turtle Geometry. You can see the finished project [on the Snap! website](http://snap.berkeley.edu/snapsource/snap.html#present:Username=hckmd&ProjectName=Turtle%20Geometry). A link to the project is also available on the *Coding in Snap! Activity* session page on the workshop website.

Once you have the project open in your browser, click the green flag to run the program, answer the Turtle's question and see what happens.When you press the space bar the pen drawn on the canvas will be erased and the Turtle will move back to its original position.

What happens when you try the following, after clicking the green flag?:

* Answer the Turtle's question with a small number (for example: 4)
* Answer the Turtle's question with a larger number (for example: 70)
* Answer the Turtle's question with a negative number (for example: -1)
* Put in a very large number (for example: 500)

## Base Project

There is a base *Snap!* project that you will make changes to for this activity. Before you move onto the next section, please make sure you have imported the base Turtle Geometry project. To download the base project, go to the *Coding in Snap! Activity* session page on the workshop website and download the *Turtle Geometry Base Project (XML file)* to somewhere easy to find on the lab computer (for example: the *Desktop*).

In Snap!, click the blank piece of paper icon (the *File* menu) and then click the *Import...* menu option. A window should open for you to select a file to import. Choose the "Turtle Geometry.xml" file you downloaded before and once it has been imported, you should see some blocks added in the *Scripts Area* and a Turtle appear in the middle of the canvas.

There are some blocks already in the base *Snap!* project. These blocks allow you to press the space bar to erase all of the pen drawings on the canvas and move the Turtle back to its original position.

The turtle sprite that is used in this project was created by the artist *Sogomn*, and has been downloaded from this website: <https://opengameart.org/content/animated-turtle>.

# Algorithms and Programs

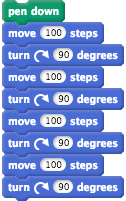
In this activity, we will take an *algorithm* and create a *Snap! program* that follows that *algorithm*. An *algorithm* is a precise and step-by-step procedure that is written for a computer to follow. *Algorithms* are often compared to recipes, as recipes also involve detailed, step-by-step instructions.

Usually when people talk about *algorithms*, they are referring to the instructions that the computer will follow. These instructions could be created in a variety of different ways. For example: written out with paper and pen or in a flowchart diagram. A *program* is an *algorithm* that has been written in a *Coding* language (for example: *Snap!*) and can be run by a computer.

For example: we may write the following steps for someone to follow (an algorithm), to draw a square:

* draw a horizontal line 10cm in length anywhere on a piece of paper
* rotate the piece of paper clockwise by 90 degrees
* draw another horizontal line 10cm in length, starting from the end of the last drawn line
* rotate the piece of paper clockwise by 90 degrees
* draw another horizontal line 10cm in length, starting from the end of the last drawn line
* rotate the piece of paper clockwise by 90 degrees
* draw another horizontal line 10cm in length, starting from the end of the last drawn line

However, these instructions are not really a *program* until they are written in *Code*, as shown in the *Snap!* blocks below:



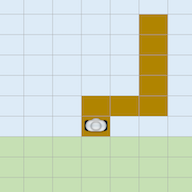
To create *programs*, you not only need to know how to create precise and step-by-step instructions that a computer can follow to complete a task (which is what we call *Computational Thinking*), but also be able to take those instructions and put them in a language that the computer can understand (which is *Coding*).

In the next section, we will look at some *Computational Thinking* concepts, that are essential for understanding how to write *algorithms*, and then use them for *Coding* our Turtle Geometry *program*.

# Sequencing

## Why is Sequencing Important?

One of the most important concepts in this activity (and *Coding* in general) is *sequencing*. Computers not only need precise instructions to complete tasks - these instructions need to be in the right order. For example, say we have a robot that we are *Coding* to navigate a jetty over water, as shown in the image below.



In the image above:

* The robot is the silver shape
* The brown squares represent the jetty
* The blue squares represent the water
* The green squares represent land

To help the robot move along the jetty, we first want the robot to move up by one square and then move to the right by one square. But what would happen if we told the robot to follow these instructions the other way around? The robot would move right by one square, then move up by one square. When we *Code* computers, they will do exactly what they are instructed to do - so in this example the robot would end up in the water!

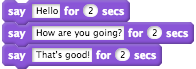
## Sequencing in Snap!

After clicking the green flag in *Snap!*, the steps in a *Snap! program* will run from top to bottom through a stack of blocks. For example, after clicking the green flag, in the stack of *Snap!* blocks below, the Turtle will:

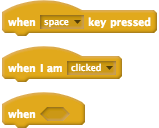
1. say *Hello* for two seconds
2. say *How are you going?* for two seconds
3. say *That's good!* for two seconds



You can also get the Turtle to do the same thing, by dragging away the "when green flag clicked" block and clicking the blocks, as shown below:



However, we do recommend that when you create a *program* in *Snap!* that you use a *when green flag clicked* block for the blocks that should be followed when the *program* starts. Sometimes, the *when green flag clicked* block is referred to as a *hat block* because it’s shaped like a hat and you may also notice that other blocks can only be snapped underneath the *when green flag clicked* block. Other examples of hat blocks in *Snap!*, which are all from the *Control* section of the *Blocks Palette*, are pictured below.



## Turtle Geometry

Now, we will start creating our Turtle Geometry *Snap! program*. During this section we will focus on how blocks in *Snap!* are run in sequence.

In the Scripts Area, add the following blocks:



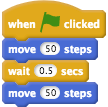
Click the green flag and see what happens.

The Turtle should move to the right in one movement. This is often unexpected - people usually expect the Turtle to move in separate steps or to do 2 movements across the screen.

The *move* blocks above do not mean that the Turtle will take 50 separate steps but that the Turtle will move 50 pixels to the right in one movement. Also, while we have two separate *move* blocks in the code above, the Turtle does not wait between these blocks. So, the Turtle moves 100 steps in total, without pausing.

Remember that when *Coding*, the instructions we give to a computer must be precise and direct. To get the Turtle to pause before the second *move* block, we must add a *wait* block.

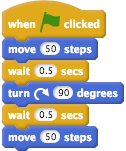
Change the code in your *program* so that it looks like this, by adding a *wait* block.



Click the green flag and see what happens.

Now the Turtle will move 50 steps to the right, pause for half a second and move another 50 steps to the right.

We want to make the Turtle move in all directions around the canvas, rather than just move to the right. To make the Turtle change direction and move, change your stack of blocks to this instead:



The *Code* above makes the Turtle move right 50 pixels, wait for half a second, turn 90 degrees clockwise (so that it is facing downwards), wait for half a second again and then move 50 pixels downwards. When we use the *turn* block, the Turtle will change its current direction by the number of degrees in the *turn* block, in the direction of the *turn block* (right/clockwise or left/counter-clockwise). When we use the *move* block the Turtle will always move the direction it is facing, unless the number of steps given is negative (in which case, the Turtle will move in the opposite direction).

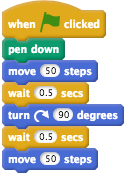
Remember that you can always move the Turtle back to its original position and make it face rightwards again by pressing the space bar.

## Drawing a Square

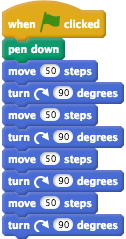
Now, we will make our Turtle draw a square. So far, we have moved the Turtle around the canvas but we have not drawn any shapes. To draw lines on the canvas, we can use the *pen down* and *pen up* blocks from the *Pen* section of the *Blocks Palette*.

The Turtle (and any *Sprite* in Snap!) can either have its pen down or up. When the pen is down and the Turtle moves around the canvas, it will draw lines that follow the *Sprite*. When the pen is up, no lines will be drawn when the Turtle moves around the canvas.

We have to tell the Turtle to put its pen down to draw a square. To do this, add the *pen down* block to your stack of blocks so that it looks like the blocks below:



Now, we are going to command the Turtle to draw a square. As briefly explained in the *Algorithms and Program* section of this tutorial, a *program* in *Snap!* that draws a square could be created with a stack of blocks like those above. Change your stack of blocks to look like the stack below, which will draw a square with sides that are 50 pixels in length.



You may notice that we no longer have *wait* blocks in the blocks and so when you click the green flag the Turtle will draw a square instantly. You can add *wait* blocks between the *move* blocks and *turn* blocks if you would like, this may also make it easier to understand all of the steps the Turtle follows. We have decided to not include *wait* blocks to keep the example snippets of *Code* concise.

If you already familiar with *Coding* you may wonder why we have not used loops to make the stack above more concise. We will explore how to use loops for *repetition* of actions in the next section.

## Checking Your Understanding

Now that we are the end of the *Sequencing* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Sequencing: Exercise 1

Look at the *Snap!* code below.



How many times will the Turtle pause before moving again? What are the total number of steps (pixels) that the Turtle will move?

### Sequencing: Exercise 2

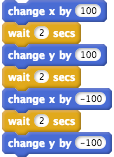
Now, the *Snap!* code is changed to this:



What will happen when the stack of blocks is clicked? Will the Turtle move to the right, then the left, the right and the left again? Or will the Turtle appear to not move at all?

### Sequencing: Exercise 2

You ask a student to explain how the Turtle moves around the canvas when it follows a stack of *Snap!* blocks. This is their *program*:



The student explains that, when the top of the stack of blocks is clicked, the Turtle will:

*Move to the right, wait for a couple of seconds, then move up a bit. It will wait again for a couple of seconds, then move to the left. It will wait again and finally move back down to where it was at the start.*

Is this explanation correct?

### Sequencing: Exercise 3

A student is creating a *program* that involves drawing equilateral triangles, that have sides 100 pixels in length, with the *pen down*, *move* and *turn* blocks. They have the stack of blocks shown below but, when they click the green flag, a shape is drawn that is not an equilateral triangle.



The student has made two mistakes in their stack of blocks above. What are these mistakes and how can the student can fix them so that the shape that is drawn, when the green flag is clicked, is an equilateral triangle?

# Repetition

## What is Repetition in Coding?

Another important *Coding* concept is *repetition*. Computers are very good at repeating actions, whether you want them to repeat something two times, one hundred times, or even to repeat something over and over forever.

There are many examples of *repetition* in real life, as well as in *Code*. For example, when you swim 10 laps in a pool, you are repeating the lap 10 times. You would also be repeating the actions that make up the stroke (for example: freestyle) over and over as you swam the lap.

Another example of *repetition* could involve repeating actions for a list of people. You may organise a birthday party and decide to write personalised emails to all of the guests. To send out all these emails, you may follow these actions until you have worked your way through the guest list:

1. Look at the next guest on the list
2. Write an email addressed to that guest
3. Go back to step 1

In *Coding*, we use loops to repeat actions. *Snap!* has three main types of loop blocks: *forever*, *repeat* and *repeat until*. Every programming language has some way of allowing *repetition* of actions, which are usually loops, but we won't worry about that in this activity.

## An Example Program that uses Repetition

The *Snap!* blocks below show the above steps (*algorithm*) as a *Snap! program*. Don't worry about trying this out in your own *Snap! program* - just read the blocks below and see how they relate to the 3 actions above. The example also includes blocks and concepts that we haven't explored. These blocks and concepts will be explained later in the tutorial, but I will also briefly explain what the blocks do in the paragraph below the blocks.



In the blocks above, the green *send email to guest's email* block is a made-up *custom block* (which we will explore in the *Functions* section of this tutorial). The *send email to guest's email* block would have *Code* to create and send an email to a party guest (which is not shown in the example above).

The *guest list* is a list of party guests' emails. We could, for example, have a list of three party guests' email addresses in this list: *bob.smith@gmail.com*, *rita.green@gmail.com*, *joan.blue@gmail.com*. The *guest number* is a variable (we will look at variables later in this activity) that changes by 1 each time the repeat loop runs its actions. The *guest's email* is another variable, which will be used to keep track of each of the *guest's email* that the invitation will be sent to.

The way this *Code* in the blocks will work is:

**Before the repeat loop:**

* The *guest number* variable will be 1

**The first time the instructions in the repeat loop are followed:**

* The actions in the *repeat loop* will run for the first time
* The *guest's email* variable will become the first item of the *guest list* (*bob.smith@gmail.com*)
* An email will then be sent to bob smith's email address
* The *guest number* will change by 1 (to 2)

**The second time the instructions in the repeat loop are followed:**

* We go back to the first block after the *repeat* block and the actions will run for the second time
* The *guest's email* variable becomes the second item of the *guest list* (*rita.green@gmail.com*)
* An email will then be sent to rita green's email address
* The *guest number* will change by 1 (to 3)

**The third (and final) time the instructions in the repeat loop are followed:**

* We go back to the first block after the *repeat* block and the actions will run for the third time
* The *guest's email* variable becomes the third item of the *guest list* (*joan.blue@gmail.com*)
* An email will then be sent to joan blue's email address
* The *guest number* will then change by 1 (to 4)

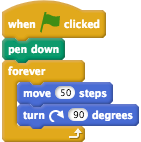
In total, the actions in the repeat block above will run three times for this example, as the number of email addresses in the guest list is three.

## Loops in Snap!

Recall that, as mentioned above, *Snap!* has three main types of loop blocks: *forever*, *repeat* and *repeat until*. In this section of the activity, we will focus on the *forever* and *repeat* blocks.

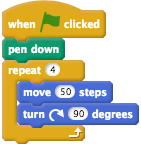
Sometimes you may have stacks of blocks in *Snap!* that are repeated in other parts of your *program*. We can use loops to simplify parts of the *program* that are repeated and consequently we can make our *Code* more concise. For example, can you see that in the stack of blocks for drawing a square there are two blocks that are repeated four times?

The *move 90 steps* block and the *turn 90 degrees* are repeated. We can change the stack of blocks so that the two repeated blocks are put in a *forever* loop, like the stack shown below:



Click the green flag and you will see that these blocks make the Turtle draw a square over and over, without stopping. This *program* will run forever and the Turtle will continually draw a square - unless we tell it to stop. You can stop the Turtle by pressing the space bar or clicking the red stop icon next to the green flag.

We don't want the Turtle to draw a square forever, and so we will use a *repeat* block instead. Change the blocks so they look like this instead:



We are now using a *repeat* loop block, instead of a *forever* loop block. The *repeat* block has a slot for a number (4 in the example above), which determines how many times the actions in the loop repeat. Try changing the 4 in the *repeat* loop block to different numbers and running the program. What happens when you use larger numbers (for example: 10). What happens when you use numbers like 0 and -1 in the *repeat* block and run the program?

## Nested Repetition

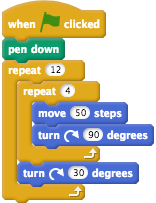
We can also put loops inside loops - this is called *nesting loops* or *nested repetition*. For example, create a separate stack of blocks to your blocks that make the Turtle draw a square, that look like this:



Click the stack of blocks and count the number of notes that play. You should hear 8 notes play. Why does this happen? The *repeat (4)* block repeats the actions inside the loop four times, these actions being the *repeat (2)* loop, which in turn plays the note 2 times. Therefore, the note is played 8 times in total. So, when you have two nested repeat loops, the actions inside the inner loop block (*play note (60 v) for (0.5) beats* in the example above) will repeat for the number in the outer *repeat* loop (4 in the example above) multipled by the number in the inner *repeat* loop (2 in the example above).

We are now going to add an extra step to our Turtle Geometry *program*. Now, we are going to change the *program* so that the Turtle will repeat the drawing of the square and change each its direction each time it does this. This will make the Turtle draw the circular pattern of squares that you would have seen when you tested the Turtle Geometry project on the *Snap!* website (in *The Project* section of this tutorial).

To do this, we will need to use *nested loops*. The program to make the Turtle dance should be changed to look this:



Now, when you click the green flag, the Turtle will follow these steps 12 times:

1. Draw a square by moving outwards from the centre
2. Turn 30 degrees clockwise

You may notice that, because the *turn 30 degrees* block is followed 12 times, the Turtle turns a full 360 degrees (1 revolution) and this is why the resulting pattern of squares is circular.

## Making the Drawn Square a Random Colour

To make the circular pattern more colourful, we can change the colour of the pen that the squares are drawn with. In this activity, we will use *Randomisation* to determine the colour of the pen.

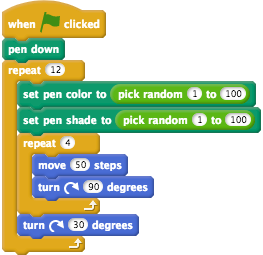
*Randomisation* is used in many different types of programs, especially ones where there is chance involved. Two examples could include a Blackjack game where *randomisation* would be used to shuffle a deck of cards and videogames where *randomisation* could be used to determine where and when enemy characters appear on the screen and how they behave. In the arcade game Pacman, for example, *randomisation* could be used to randomly choose which direction one of the ghosts chasing Pacman would move next.

In *Snap!* you can create random numbers with the *pick random* block, which looks like the block shown below:



If you find the block in the *Blocks Palette* (in the *Operators* section) and click on it a few times, speech bubbles with a random number between 1 and 10 will appear. The *pick random* block has 2 slots - the first is for the lower bound (1 by default) and the second is for the upper bound (10 by default). The random number that is generated will be in this range (1 to 10, in the example block above).

We can combine the *pick random* block with the *set pen color to* block and the *set pen shade to* block (which are both in the *Pen* section of the *Blocks Palette*) to make the squares a random colour. The stack of blocks can now be changed to look these shown below:



The range is 1 to 100 for both *pen* blocks because 1-100 is the scale that Snap! uses for both colours and shade.

## Checking your Understanding

Now that we are the end of the *Repetition* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Repetition: Exercise 1

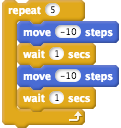
Look at the blocks below.



How many times will the note play?

### Repetition: Exercise 2

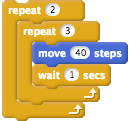
You ask a student to explain what the Turtle will do when you click the stacks below.



The student says that the Turtle will move to the right 5 times when you click on the stack of the blocks. This is an incorrect explanation of the above blocks for two reasons, why is that and what will actually happen?

### Repetition: Exercise 3

Look at the blocks below.



How many times will the Turtle move to the right? How many times does the outer loop repeat and how many times does the inner loop repeat?

# Variables

## What are Variables?

In *Snap!*, and *Coding* in general, a *variable* is a name for a value (for example: a number like 5) that can change. For example, look at the blocks in the below image.



In the example above, we have a *variable* called *number* that changes as we move through down the stack of blocks. What happens when you click this stack of blocks?

1. The *number* variable's value is set to 2.
2. The Turtle will say the value of *number* (2 at this point) for 2 seconds
3. The *number* variable's value is then set to 4
4. The Turtle will say the value of *number* (4 at this point) for 2 seconds

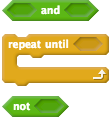
*Variables* are an essential *Coding* concept that are used in most programs. For example, if we made a basketball game in *Snap!* and wanted to keep track of each team's score, we would need to use two *variables*: 1 number *variable* for each of the teams. These *variables* (let's call them *Home Team Score* and *Away Team Score*) would start with a value of 0 and then every time one of the teams scored a basket, the value of the appropriate *variable* would be increased by 2.

There are different types of *variables* in different *Coding* languages and sometimes *variables* work differently in other languages. However, in this activity we will focus on how they work in *Snap!*.

*Variables* can take on four different types of values in *Snap!*:

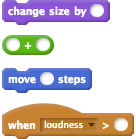
1. **Boolean**: the value of the *variable* is either *true* or *false*
2. **Number**: the value of the *variable* is an integer (a whole number like *2* or *3*) or a decimal number (for example: *1.34* or *2.5*)
3. **String**: the value of the *variable* is a piece of text (for example: *Hello* or *Dan*)
4. **List**: the value of the *variable* is a collection of other values. In *Snap!*, Lists can contain values and *variables* of any other type (including other *Lists*). For example, you could have a *List* of students' marks (for example: [*99*, *97*, *85*] would be a *List* of three *Numbers*). *Lists* will be explained in more detail towards the end of this activity.

Knowing the different types of *variables* in *Snap!* is important because the different blocks in *Scratch* expect *variables* with specific types of values. For example, the blocks below expect a *variable* with a *Boolean* value to be placed in the empty slot/s:



Notice that in the above blocks, the slots where the blocks can be placed are shaped the same.

In the next example below, the blocks pictured expect *variables* with *Number* values to be placed in the empty slot/s.



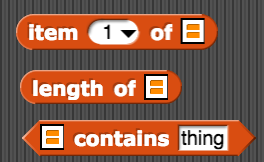
Notice that, like the *Boolean* value example, the empty slots are the same shape as each other (although they are more circular than the *Boolean* values slots).

In the next example below, the blocks expect a *variable* with a *String* (text) value:



These all have the same shaped empty slots as well. However, for *String* values, the empty slots are rectangular shaped.

Lastly, blocks that expect a *variable* that is a *List* all have an orange and white *List* icon in them:



Knowing the type of values that should be used in the different blocks can be important when trying to find out why *Code* is not doing what you expect to do. For example, read the blocks in the stack below, which involves three different *variables* (*Introduction*, *Name* and *Result*):



This could be an example of some *Code* that a student puts together that is supposed to have the Turtle introduce itself. You might expect that, because we are adding *"Hello. my name is "* and *"Dan"*, that the value of the *Result* variable would be: *"Hello, my name is Dan"*. Instead, because the add block expects *Number* values - not *String* values, the value of the *Result* variable is actually *0*. To make the sentence *"Hello, my name is Dan"* with the *Introduction* and *Name* variables we would use the *join* block instead, as shown below:



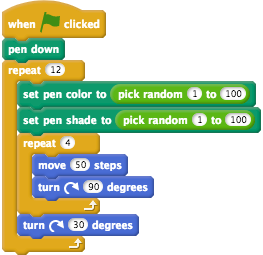
## Adding a Step Counter to Our Program

Let's imagine that our Turtle wants to know how many movements it makes when it is drawing the squares for us. This would work similarly to a pedometer that keeps track of how many steps you take each day. We will create a *variable* called *number of movements* that will keep track of how many movements the Turtle makes.

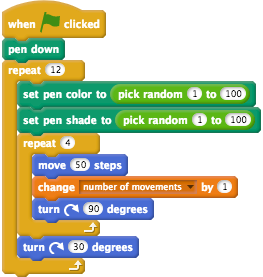
To create the *variable*, follow these steps:

1. Go to the *Variables* blocks section of the *Blocks Palette*.
2. Click the *Make a Variable* button
3. Name the variable *number of movements*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

We want to add 1 to this *variable* every time the Turtle moves, using the *change variable* block. Our Turtle Geometry program is shown below. How many *change variable* blocks should be added below and where do you think those blocks should be placed?



The *change variable* block should be added to your Turtle Geometry program and placed after the *move* block, as shown below:



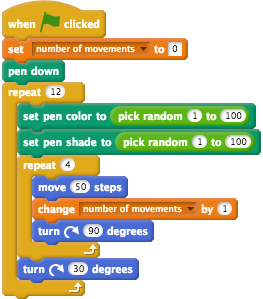
Click the green flag and watch the *number of movements* variable's Stage monitor (the box on the canvas that has the text: *Turtle number of movements*). Notice that the *number of movements* *variable* increases is 4 times 12 because the Turtle takes 4 movements to draw a square and it draws 12 squares in total.

## Resetting the Movement Counter

If you click the green flag a few times, you will notice that the *number of movements* keeps increasing. For example, if you have clicked the green flag three times, the *number of movements* *variable* will be 144 (48 times 3).

We now want to reset the movement counter at the beginning of our program. Setting the value of a *variable* at the beginning of a program is called *initialisation*, which is good *Coding* practice. Another common example of *initialisation* is that when you play a video game and it ends, when you start a new game your score will be reset to 0.

To reset our movement counter, we will add a *set variable* block after the *when green flag clicked* block, as shown below:



Now, run the Turtle Geometry *program* a few times and you will notice that each time the *number of movements* *variable* will start at 0 and finish at 48.

## Repeating using a Variable

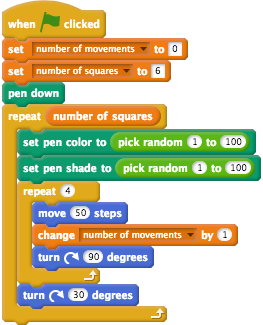
*Variables* can be placed in any of the blocks that have an empty slot. For example, we can use a *variable* in the *repeat* block, which will determine how many times the actions in that *repeat* block are repeated. In this part of the activity, we will create a *variable* called *number of squares*

Create a variable called *number of squares* by following these steps:

1. Go to the *Variables* blocks section of the *Block Palette*
2. Click the *Make a Variable* button
3. Name the variable *number of squares*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

To use our *number of squares* variable in our *repeat* loop, we will need to add two blocks. The first block will be a *set variable* block, where we will set the *number of squares* *variable* to the number of squares we want the turtle to draw in its circular pattern. The second block will be the *number of squares variable*, which can be found in the *Variables* section of the *Blocks Palette*.

You can see the stack of blocks with these two blocks added below:



Change your *program* so that it includes these blocks and matches the stack above. Notice that if you change the *6* in the *set number of squares* block and click the green flag, the Turtle will draw that number of squares (unless you use 0 or a negative number).

## Using a Variable in an Operator Block

If you tried using different values for the *number of squares* variable, you may have noticed that some numbers would result in a pattern that was not completely circular or that the Turtle would draw over the top of its pattern. Numbers smaller than 12, for example: 6, result in an unfinished circular pattern and numbers larger than 12, for example: 20, cause the Turtle continue moving and drawing squares after it has drawn the circular pattern.

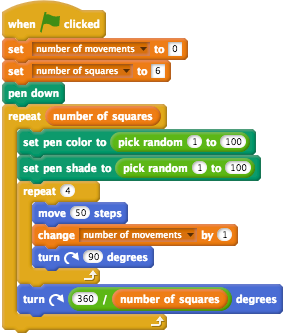
How could we change the *program* so that the turtle always draws a circular pattern of squares, regardless of the value of the *number of squares variable*?

We can use *Variables* and one of the *Operators* blocks to do this, as a full revolution will always be 360 degrees. Consequently, we know that the degrees the Turtle should turn after every drawn square should be 360 divided by the *number of squares variable*. For example, if the Turtle draws 20 squares, it should turn 18 degrees after drawing every square because 18 times 20 is 360.

The *Operators* block we will use, along with the *number of squares* variable, is the division block which looks like this:



Change your program so that it looks like the *Code* below:



Try with different values for the *number of squares variable* and see if the Turtle always draws a circular pattern.

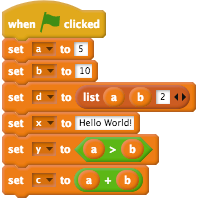
In the next section, after the *Check Your Understanding* questions, we will look at *Functions*.

## Check Your Understanding

Now that we are the end of the *Variables* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Variables: Exercise 1

Have a read of the code blocks below:



There are six *variables* in the code blocks above. What are they named, what type (*Boolean*, *Number*, *String* or *List*) are they and what will their values be after the last block?

### Variables: Exercise 2

Read the blocks below, which involve a *variable* (*times*) that is used in a *repeat* block.



What is the value of the *times variable* after the last block in the stack above? How many times does the note play?

### Variables: Exercise 3

A student has created a short *program* as part of a bigger game project. They currently have the following blocks:



The student wants to change the blocks so that every time they press the space bar, the value of the *time space bar pressed* variable decreases by 1. How could they change the blocks above to do that?

### Variables: Exercise 4

You have decided to use the blocks from the example given in the *repetition* section, where you had a guest list and where sending invitation emails to your guests. Let's say that you tried to remember the blocks from the previous example and ended up with the following stack of blocks:



You send out the invitations using this *Snap!* program but you only recieve RSVPs from half of the invited guests. Why could this be?

# Functions

## What are Functions?

*Functions* in *Coding* are similar to *Functions* in Mathematics, but there are some some important differences between them. Like in Mathematics, *Functions* can be given some *input* and can then return some *output*. For example, in Mathematics when the function *f(x) = 2x* is given the *input* 4, it will always *output* 8.

Unlike Mathematics, it is not mandatory for *Functions* in *Coding* to have *inputs* or *outputs*. For example, in *Snap!* the *show* and *hide* blocks (in the *Looks* section of the *Blocks Palette*) have no *inputs* or *outputs*. Also, *Functions* in *Coding* do not always *output* the same result for the same *input*, which is not true for *Functions* in Mathematics. For example, when we used the *pick random 1 to 10* block in the last section we would receive a different result every time we clicked the block, even though the inputs (1 and 10) did not change.

*Functions* are mainly used to help make *Code* more concise, easier to maintain and to cut down on repetition. In *Snap!* functions can be defined by making your own blocks, which will appear in the *Blocks Palette*. For example, if we wanted to copy our stack of blocks that draws the circular pattern of squares to somewhere else in our *program*, we can combine these blocks into one block. This could make it easier for someone reading your stack of blocks (because there are less blocks to read) and it also allows you to make changes to the *algorithm* for drawing a square in just one place.

In comparison to *Scratch* (and other *blocks languages*), the design of *Snap!* allows for much more powerful and flexible ways of designing and using *Functions*. For example, while *Scratch* allows you to define your blocks, you can only create blocks of one of the three types (*Command*, *Reporter* and *Predicate*, which are explained below) that you can with *Snap!*.

We will not go into detail about the advanced features of blocks in *Snap!* (for example: *recursion* or *higher-order functions*) in this activity. If you are interested in finding out about more advanced uses of blocks in *Snap!*, we recommend working through the *Beauty and Joy of Computing Curriculum's* course materials (the link for the materials are available on the workshop website).

## Types of Blocks in Snap!

When making a block, you can choose which category it fits into (for example: *Pen* or *Operators*). The category chosen will affect the colour of the block and determine which category menu in the *Blocks Palette* it will appear in. There are also three types of blocks that you can make in *Snap!*. The key difference between these types of blocks is the type of *output* value of the block.

### Command Blocks

A *Command* block has no *output* at all. The instructions in the *Command* block will be followed but there will be no *output*. For example, if you click the *move 10 steps* in the *Motion* section of the *Blocks Palette*, the Turtle move 10 steps across the canvas but, unlike *Reporter* and *Predicate* blocks, there will be no *output* message that appears in a speech bubble.

Examples of *Command* blocks are shown in the image below. You may notice that they are all shaped like each other.



### Reporter Blocks

*Reporter* blocks are blocks that have some *output*, such as a *Number* or *String*. For example, if you click the *tempo* block in the *Sounds* section of the *Blocks Palette*, a speech bubble with a number (likely *60*) will appear from the clicked block.

Three different examples of *Reporter* blocks are shown below.



As you can should see, these blocks are shaped similar to each other but they are also more circular than the *Command* blocks.

### Predicate Blocks

*Predicate* blocks are similar to *Reporter* blocks because both types of blocks *output* some value. However, the *output* for a *Predicate* block has to be a *Boolean* value (*true* or *false*). For example, output of the *key space pressed* block in the *Sensing* section of the *Blocks Palette* will be either *true* (if the space key is pressed) or *false* (if the space key is not pressed).

A couple of examples of different *Predicate* blocks are shown below:



You should notice that the *Predicate* blocks are also shaped similar to each other.

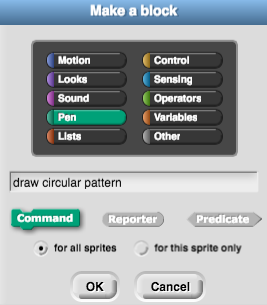
It is important to understand the different types of blocks and *output* values, because this could help you understand why students' *Code* is not working as expected. For example, only *Predicate* blocks and *Boolean* values can be used in the empty slots of some *Snap!* blocks, such as *if* blocks.

## Defining a Block

Now, we will define a block to command the Turtle to draw a circular pattern with a specified number of squares. This will involve moving the *Code* we have for drawing a pattern into a block that we will title *draw circular pattern*.

To define a block, you can right-click anywhere on the *Scripts Area* (except for the stacks of blocks) and click the *make a block...* option in the menu. You could also click one of the *Make a block* buttons in the *Blocks Palette*, but we recommend right-clicking on the *Scripts Area* for the following steps.

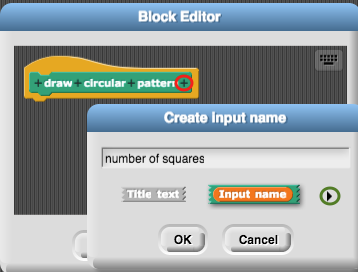
After selecting the *make a block...* option from the menu, a popup window titled *Make a block* will appear. We want the *draw circular pattern* block to appear in the *Pen* section of the *Blocks Palette*, so you should select the *Pen* option in the *Make a block* window. You will also have to enter the block name (*draw circular pattern*) and make sure that the *Command* block is selected. The form in the *Make a block* popup should look like the image below:



After clicking the *OK* button, the *Block Editor* will open. The *Block Editor* window is where you will add *Code* blocks to define the actions that your new block will perform. You can re-open this window at any time by right-clicking on one of the blocks you have made and clicking the *edit...* option.

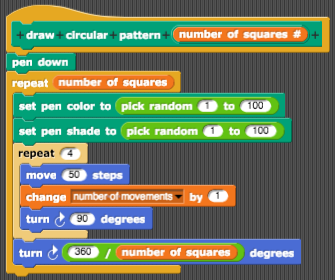
We now want to add an *input* to the *draw circular pattern*, which will be used to determine how many squares will be drawn in the circular pattern. To add an *input* to the block:

* Click the + next to *pattern* in the block name (as highlighted by the red circle in the image below)
* When the *Create input name* window appears, put the name of the input (which we will call *number of squares*) in the textbox as shown in the image below

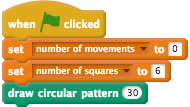


Before you click *OK*, click the right arrow (highlighted by the green circle in the image above) and you will see options appear for choosing the type of *variable* that the *input* is (such as: *Number*, *Boolean* and *List*). We want the *number of squares* to be a *Number*, so select the *Number* option from these options and click *OK*. The *number of squares* input should appear in an orange oval (like a *variable*) in the block name and there should also be a # added to the end (to signify that the input is a number).

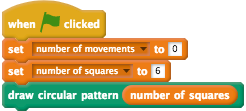
Next, drag your stack of blocks that draws the pattern (from the *pen down* block to the bottom of the stack) and place it under the *draw circular pattern* hat block in the *Block Editor* window. The blocks in your *Blocks Editor* window should now look like the image below:



You can now change your stack of blocks to command the Turtle to draw the circular pattern by adding the *draw circular pattern* block to the program. You can put any number that you like as *input* to the *draw circular pattern* block. We have used 30 as the *input* in the example below.



If you change the number (30) that is used for *input* for the *draw circular pattern* block and click the green flag, you will notice that the Turtle draws a circular pattern with that number of squares. Next, instead of changing the number in the *draw circular pattern*, we are going to use the *number of squares variable* as *input* to this block. To do this, change your blocks so they look like the stack of blocks shown in the below image.



If you change the value of the *number of squares variable* and run the click the green flag, the Turtle will draw the number of squares in the *set* block.

## Check Your Understanding

Now that we are the end of the *Functions* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Functions: Exercise 1

A student creates a block called *is Sprite moving?* that they want to have an output of *true* if a Sprite is moving on the canvas and *false* if the Sprite is not moving. The block is in the *Motion* section of the *Blocks Palette* and looks like the image below.



The student is trying to put the block in an *if* block but it does not fit in the empty slot. What type of block (*Command*, *Reporter* or *Predicate*) is the *if Sprite moving?* block and what type of block should it be changed to so that it fits into an *if* block?

### Functions: Exercise 2

A made-up block from the *Motion* section of the *Blocks Palette* is shown in the image below.



How many *inputs* does the block above have and what type of *variables* are the inputs?

# User Input

## What is User Input?

*User input* is a term that means that the person using the *program* (the user) interacts with the *program*, usually by (but not always) using a keyboard/mouse.

For example, you might register for an event (for example: a workshop) using an online form. The *user input* in this example would be the details you would enter into the form (such as your name and email address). Another example of *user input* is controlling a character in a videogame with a joystick. The videogame would have some *Code* that would take that input from the joystick (for example: moving the joystick to the right) and then perform some action (in this example: moving the character to the right).

In *Snap!*, there a few different ways to get *user input*. Examples of *user input* in *Snap!* include:

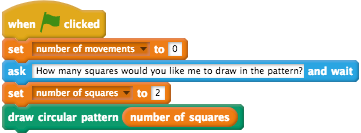
* Asking the user a question with the *ask and wait* block in the *Sensing* section.
* Detecting when a user has pressed a key, with the *when key pressed* block in the *Control* section and *if key pressed?* block in the *Sensing* section.
* Detecting when a user has clicked on a sprite (for example: when a user clicks on the Turtle) with the *when I am clicked* block.

In this section of the activity we will focus on the first example: asking a question.

## Adding a Question to Our Program

We are going to make the Turtle ask the user how many coloured squares should be in the circular pattern it draws.

The Turtle will need to ask this question before the *draw circular pattern* block. The stack of blocks below include the *ask and wait* block.



To make your stack of blocks match the above image, make sure that you:

* Drag an *ask and wait* block from the *Sensing* section of the *Blocks Palette* and place it after the *set number of movements to 0* block.
* Replace the text in the *ask and wait* block with: *How many squares would you like me to draw in the pattern?*

As we already have a *variable* - called *number of squares* - that is used to determine how many squares the Turtle will draw, we will now set that *variable's* value to the answer given by the user. For example, if the user answers the Turtle's question with 2 - we want the value of the *number of squares variable* to become 2.

There is a special block in *Snap!* called *answer*, which you can find in the *Sensing* section of the *Blocks Palette*. When we use an *ask and wait* block, the *answer variable* will be set to the value given by the user to this question. For example, read the stack of blocks below:



Say that you click the stack of blocks above and the Turtle asks: *What day of the week is it?* If you then typed in the name of a day, for example: *Monday*, the Turtle would then say: *Yes! It is Monday*

Now, we will use the answer given to the Turtle (the *user input*) to set the value of the *number of squares variable*. To do this, drag the *answer* block from the *Sensing* section and replace the 6 with the *answer block*. Your *set number of squares* block should now look like this:



Click the green flag button, answer the Turtle's question and watch the Turtle draw the squares. Does the Turtle draw the number of squares you expect?

After the Turtle has finished its drawing, the value of the *number of movements* should be: the value of the *number of squares* variable multiplied by 4.

## Check Your Understanding

Now that we are the end of the *User Input* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### User Input: Exercise 1

A student has written the following *Code*:



What will happen when you type *John* in the answer box and press enter?

### User Input: Exercise 2

A student has the following stack of blocks, which are part of a bigger game project. They say that the *Code* is not working properly but they don't know why.



The student also shows you an example of what is confusing them:

* The student clicks the green flag and the *program* starts
* The student enters their name in when the Cat asks their name (for example: *Dan*) and presses enter
* The Turtle says: New player = *Dan* and the *program* ends
* The student asks their friend (who is named *Ben*) if they want to play the game
* Ben clicks the green flag and the Turtle says: New player = *Dan*, then asks: What's your name?

The student wants the Turtle to ask the player's name then say: New player = *player name* but the *program* does not work this way. What do you think is causing this problem to happen and how it can be made to work as the student expects it to?

### User Input: Exercise 3

Another student is creating a game where the user has to navigate the Turtle through a maze. They are currently working on the *user input Code*, shown in the blocks below, and are finding that when they try and move the Turtle with the arrow keys, the Turtle is not moving as the student expects.



What will they need to change to get the Turtle to move as expected? Hint: when the user presses the right arrow key, will the Turtle move to right? Why or why not?

# Branching

## What is Branching?

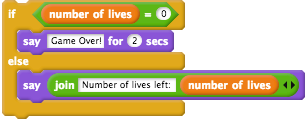
*Branching* in *Coding* refers to the computer following different instructions based on certain conditions. *Branching* is also sometimes called *selection*.

There are many examples of *branching* in real life. For example, say that you are hosting a barbecue and a friend has offered to buy the bread from the supermarket. You have heard that there has been a shortage of loaves of bread in the local area and so you give them the following instructions:

* If there are loaves of bread at the supermarket, buy a loaf of bread
* If there's no loaves of bread at the supermarket, buy 3 packs of breadrolls instead

This example is *branching* because, depending on the condition (the availability of loaves of bread), different instructions are followed.

In *Snap!*, the main blocks to use for branching are the *if then* and *if then else* blocks. There are also many examples of *branching* in *Snap!* programs, as they are an essential *Coding concept*. One example of *branching* in *Snap!*, that might be part of a bigger *Snap!* project, could be blocks that say different messages depending on how many lives a character has left in a video game.



This stack of blocks above could run every time a character loses a life (for example: by falling in water) and the *number of lives variable* is consequently decreased by 1. When the character has 0 lives left, the message: *Game Over!* will appear. Otherwise, the message about the *number of lives* left will appear.

## Dealing with Negative Numbers

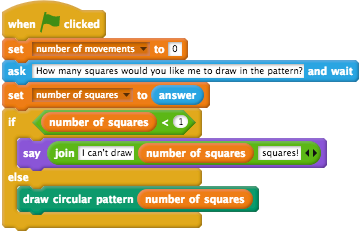
We will use *branching* to deal with what is called *invalid input*. *Invalid input* is *user input* that is nonsensical or that causes the program to do unexpected things. In this example, we will look at a specific *invalid input*: answering the Turtle's question about how many squares it should draw in the circular pattern.

What happens if we answer the Turtle's question (*How many squares would you like me to draw in the pattern?*) with a negative number (for example: -2) or 0?

If you try it and see what happens, you may notice that the Turtle doesn't do anything. This is because the Turtle can't draw a negative number of squares or 0 squares.

We will add *branching* to our Turtle Geometry program, so that the Turtle tells the user that they need to input a positive number.

Look at the *Snap!* blocks below, where an *if then else* block, a *say* block and a *join* block have been added to the *program*:



Change your Turtle Geometry *program* to match the above stack of blocks. Once you have made these changes, answer the Turtle's question with a number like 0 or -2. The Turtle should say that it cannot draw that many squares and refuse to move around the canvas.

If you enter a number larger than 0, such as 10, the Turtle will follow the instructions (drawing the circular pattern of squares) that are inside the *else* block.

## Adding an Upper Limit to the Squares Drawn

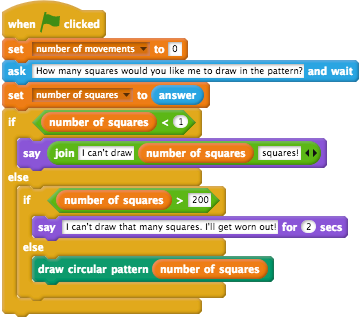
Now, let's say that the Turtle will not only refuse to draw 0 or a negative number of squares, but that it will also refuse to draw a pattern with more than 200 squares - because if it draws this many squares it will get worn out!

To add this upper limit we are going to have to use *nested branching*. *Nested branching* in *Snap!* involves having *if then* and/or *if then else* blocks inside of other *if then* and/or *if then else* blocks. This is similar to the *nested loops* we looked at earlier, where we have *repeat* blocks inside other *repeat* blocks.

We want to change our Turtle Geometry *program* so that the following happens:

* The Turtle asks how many squares you want it to draw in the pattern
* We check if the user answers with 0 or a negative number
  + If the answer is 0 or a negative number, the Turtle tells the user that it cannot draw that many squares and the *program* ends
  + Otherwise (if the answer is positive and not 0), we then check if the answer is larger than 200
    - If the answer is larger than 200, the Turtle tells the user it cannot draw that many times and the *program* ends
    - Otherwise, the answer is not larger than 200, the Turtle then draws a circular pattern with the number of squares the user answered with and then the *program* ends

To turn this *algorithm* into *program code* we need to add another *if else then* block, as shown in the blocks below:



Have a look at the bullet points above that describe the *algorithm*, the *Snap!* blocks and compare the two. Can you see how these are related to each other, and how the different instructions will be followed based on the *user input*?

## Check Your Understanding

Now that we are the end of the *Branching* section, try and answer the questions below. If you need any clarification about these questions, please let us to know. The solutions to these questions are also available on the workshop website.

### Branching: Exercise 1

Look at the *program* below that asks the user to enter a secret password, in order to get access to some secret documents.



What will happen when the user answers the question (*What's the secret password?*) with: *12345*?

### Branching: Exercise 2

A student creates a game and decides that only people aged 5 to 95 are allowed to play it. They create the following stack of blocks.



What will happen if the game player answers the question (*What's your age?*) with 10? What will happen if the player is aged 4? Or 96?

The student also notices that, regardless of what age the player is, the message: *Let's start the game!* appears. The student only wants this message to appear if the player is the right age (aged between 5 and 95) to play the game. To get the blocks to work this way, they will have to move the last *say* block. Where should the *say* block be moved?

### Branching: Exercise 3

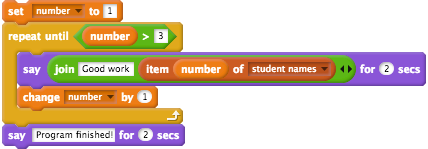
You have a list of 30 student names that has been sorted by their scores in a recent quiz, in descending order. The first student name is the name of the student who scored the highest mark in the quiz. The second student name is the name of the student who scored the second highest mark in the quiz, and so on.

You would like to create a *program* where the Turtle congratulates the students who scored the three highest marks in the quiz. The three highest-scoring students' names are: *Sophie*, *Tim*, and *Mary*, and consequently they are the first three names in our *student names* list variable in the blocks below.

You find out about the *repeat until* block, which you think you will be able to use to create this program. The *repeat until* block combines a few of the *Coding* concepts in this activity, including *Repetition* and *Branching*.

Unlike the *forever* and *repeat blocks* however, actions inside a *repeat until* block repeat until some condition becomes true. An example of repeating an action until a condition is true in your everyday life is eating a meal until you're full. When eating cereal, you may repeat the same action (for example: eating a spoonfull of cereal) until you are full. Another example would be swimming laps in a pool until you are too tired to continue swimming.

To create the program where the Turtle congratulates the top three students, you create the following stack of blocks:



Can you explain, step-by-step, what happens when these blocks run with a *List variable* called *student names*? The first three students in the *student names* list are: *Sophie*, *Tim* and *Mary*.

# Lists

## What are Lists?

A *List* in *Snap!* is a a collection of values (such as: *Numbers*, *Booleans* or *Lists*). For example, you may have a *List* of *Numbers*: 1, 2, 3, 4, 5. Most *programming* languages have *Lists* or similar types of collections (for example: the *R* language has *Vectors* and the *MATLAB* language has *Arrays*).

*Lists* can contain values of any types and they do not all have to be the same type. For example, a *List* in *Snap!* can contain a mix of *Numbers* and *Strings*. *Lists* can also contain other *Lists*, which makes it possible to create tables of data and matrices in *Snap!*.

Some examples of different blocks that create and manipulate *Lists*, which are from the *Variables* section of the *Blocks Palette*, are shown in the image below.



* The first block in the image above creates a new empty *List* (notice that there are no empty slots in the block). An item could be added to the *List* by clicking the right-facing arrow.
* The second block in the image above sets the *value* of the *numbers variable* to a *List* of 3 numbers (1, 2 and 3).
* The third block in the image above sets the *value* of the *mixed variable* to a *List* that has 3 items that are different types. These items are *Hello* (a *String*), 2 (a *Number*) and a *List* that contains one item - the number 3.
* The fourth block in the image above adds an item (4) to the *numbers List*. After clicking that block the *numbers variable* would be a *List* of 4 numbers (1, 2, 3 and 4).

Unlike *Scratch*, *Lists* in *Snap!* can be used as *input* to blocks that you define yourself and *Lists* can also be *anonymous* (have no name). These differences mean that you can use *Lists* in *Snap!* in much more advanced and flexible ways than in *Scratch* (for example: *mapping* a *function* over a *List* or *filtering* a *List* of numbers). We will not explain these advanced uses of *Lists* in *Snap!* in this activity, but you can find out more about how *Lists* in *Snap!* work by referring to the *Beauty and Joy of Computing Curriculum's* course materials (the link for the materials are available on the workshop website).

## Adding Items to a List

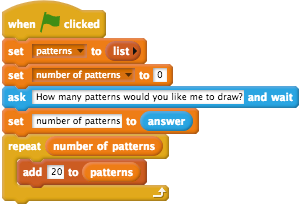
We are now going to change our Turtle Geometry *program* so that the Turtle draws multiple patterns on the canvas. This is going to involve combining *Lists* with two other concepts that you learned about earlier in this activity (*Repetition* and *User Input*).

Firstly, we will have the Turtle ask the user how many patterns you would like it to draw. After asking this question, we will create a *List* called *patterns*. This *List* will be a collection of numbers that determine the patterns that the turtle draws. For example: if the *patterns* has the items 40 and 80, then the turtle will draw 2 circular patterns: one with 40 squares and the other with 80 squares.

Create a new *variable* called *number of patterns* by following these steps:

1. Go to the *Variables* blocks section of the *Blocks Palette*.
2. Click the *Make a Variable* button
3. Name the variable *number of patterns*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

Now, repeat the same steps above for another new *variable* called *patterns*. The image below shows how the code can be changed to make the *patterns* list the appropriate size. Note that the rest of the blocks from our existing Turtle Geometry *program*, which follow the blocks shown below, have not been shown for brevity.



Click the green flag and answer the Turtle's question. You should see the *Turtle patterns* Stage monitor on the canvas, which will show the items that are in the *List*. At the moment, the blocks shown above create a *List* and then adds a new item (20) for the amount of times that the user enters for the *number of patterns variable*. This does not affect how many patterns the Turtle draws yet, however, the Turtle will still only draw one pattern. In the next steps, we will use *iteration* to draw multiple, circular patterns.

## Iterating Through a List

You will learn about how to iterate through a *List* of items in this part of the activity.

You may have noticed that the Stage monitor for the *patterns List* (which will look like the image shown below) has a number to the left of each item in black (1 and 2 in the image below).



In *Coding* that number to the left is called the *index* and this indicates where in the *List* the item is. The first item in the *List* has an *index* of 1, the second item of the *List* has an *index* of 2, and so on. You can use the *item of* block (in the *Variables* of the *Blocks Palette*) to. For example, the blocks below show how to use the *item of* block.



When you click on the stack of blocks below, the Turtle would:

1. say *Dan* for 2 seconds, as *Dan* is item 2 of the *names List*.
2. say *Tim* for 2 seconds, as the *Tim Index variable* will be 1 and *Tim* is item 1 of the *names List*.
3. say *Ben* for 2 seconds, as the length of *names* will be 3 and *Ben* is item 3 of the *names List*.

It is common to use a *variable* (that is normally called *i*) to move through a *List* and repeat an action for each item in the *List*. This process is often referred to as *iterating through a List*.

We can *iterate through a List* in *Snap!* by following this algorithm:

1. Create a *variable* called *i* and set it to 1
2. Start a loop that repeats for the length of the *List*
3. Perform some action on item *i* of the *List* (for example: drawing a pattern with a number in the *patterns* list)
4. Increase the *i* variable by 1
5. Repeat steps 3 and 4 until *i* is the length of the *List*

We will use *iteration* now, to make the Turtle ask how many squares the *user* would like in each of their patterns.

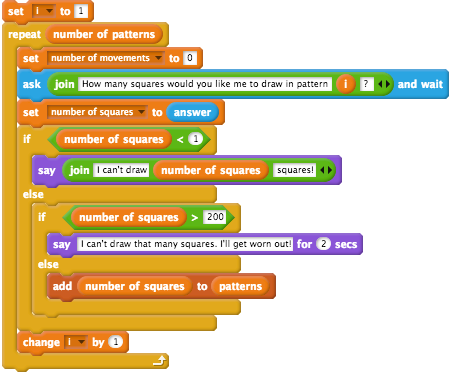
First, create a new *variable* called *i* by following these steps:

1. Go to the *Variables* blocks section of the *Blocks Palette*.
2. Click the *Make a Variable* button
3. Name the variable *i*
4. Tick the *For this sprite only* checkbox
5. Click the *OK button*

Drag the *draw circular pattern* block away from the stack for now, we will use it later. In the space where you dragged the *draw circular pattern* from, place the following blocks instead:



We are going to make four changes to the *program*. The stack of blocks after these changes have been made is shown below. Note that the blocks from the *repeat number of patterns* upwards are not shown for brevity. The details about each of the changes made to the *program* are explained after the image.



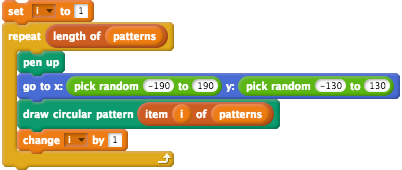
The changes to the *program* are:

1. The *add [20] to (patterns)* block has been removed from inside the *repeat number of patterns* block.
2. A *set* block, that makes the value of the *i variable* 1, has been added before the *repeat number of patterns* block.
3. The stack of blocks that followed the *repeat number of patterns* block has been placed inside that *repeat* block.
4. A *join* block has been added inside the *ask* block, which makes the Turtle ask about the number of squares for each pattern.
5. Lastly, a *change i by 1* block is added, to increase the *i variable* every time the *repeat* block repeats

Now, if you click the green flag and answer the Turtle's question about the number of patterns, it will then ask you for the number of squares you would like it to draw in each of those patterns.

## Drawing the Patterns at Random Spots on the Canvas

Finally, we are going to use *iteration* again to make the Turtle draw the patterns on different spots on the canvas. After the *repeat number of patterns* block, you should add these blocks:



Now, when you run the program, the Turtle will:

1. Ask how many circular patterns you would like it to draw.
2. Ask how many squares you would like in each pattern.
3. For each pattern, it will move to a random spot on the canvas and draw the pattern.

The ranges chosen for the *pick random* blocks should stop the Turtle from drawing outside of the canvas' boundaries. The ranges for the random x coordinate has been chosen because -190 is negative 240 plus 50 and 190 is 240 minus 50 (the canvas' width is 480 pixels). Similarly, the ranges for the random y coordinate has been chosen because -130 is negative 180 plus 50 and 130 is 180 minus 50 (the canvas' height is 360 pixels).

If you find that the Turtle Geometry *program* is not working as you would expect it to, you can view an image that shows what the final Turtle Geometry *program* should look like on this session's *Solutions* page. You can compare the image on the workshop website with your *program* and see if there are any differences.

## Check Your Understanding

### Lists: Exercise 1

There are three *Lists* (*a*, *b* and *c*) in the stack of blocks shown below.



What is the length of each of the different *Lists*?

### Lists: Exercise 2

After the following stack of blocks is clicked, what will the Stage monitor for the *numbers variable* look like?



# Conclusion

You have now reached the end of this activity - good work!

You have learned about the following fundamental *Coding* concepts in this activity:

* *Algorithms and Programs*
* *Sequencing*
* *Repetition*
* *Variables*
* *Functions*
* *User Input*
* *Branching*
* *Lists*

While the *program* that we have created is relatively simple (in comparison to a video game like *Space Invaders* or *Tetris*, for example) you can see that all of the concepts listed above were involved in the design of the *program*. As you learn more about *Coding* and see more *programs*, you are likely to see every one of these concepts present in most *programs*.

It is important to remember and understand these concepts because they are foundational to *Coding* and *Computational Thinking*. You may also notice that these are the terms used through the national curriculum's [Digital Technologies content descriptions](http://v7-5.australiancurriculum.edu.au/technologies/digital-technologies/curriculum/f-10?layout=1) and we hope that learning from this activity helps you when you read these.

If you would like to work through this activity again to practice these concepts or to refresh your memory, this activity will be available indefinitely on the workshop website. We have also created a glossary webpage on the workshop website, that you can use to review some of the meaning of the terms introduced in this activity.