

Logo Blocks Tutorial ~ Advanced Mode

If you haven't use Logo Blocks before, start with the [Beginner Mode tutorial](#).
Next, go through the [Intermediate Mode tutorial](#).
Finally, return here to learn even more block commands!

When Logo Blocks first starts up, you see the **Beginner** set of blocks. Click the word **Advanced** to get to more block commands.

In **Advanced** Mode, blocks are organized into twelve categories. Click on any word in the list at the left to get to the blocks in that category.

Moves	Moves: turn and move the turtle, set the speed, set the screen mode
Turtles	Turtles: set the turtle shape and size, hide or show it, use many turtles
Draw	Draw: set colors and pen size, pen up or down, draw shapes, fill, erase
Sound	Sound: say words or sounds
Events	Events: act on certain events that occur
Flow	Flow: use blocks that control what happens (IF...THEN and more)
Math	Math: use random numbers and math operations
Words and Lists	Words and Lists: put together and take apart sets of letters and numbers
Input	Input: work with all types of data you can give to blocks
Output	Output: print text, capture user input
Procedures	Procedures: write simple procedures with or without an input
Variables	Variables: create and use variables to create flexible procedures

All of the **Intermediate** mode are also in **Advanced** mode and this tutorial won't cover them. Refer to the [Intermediate mode tutorial](#) for details about using blocks in these categories:

- **Moves**
- **Turtles**
- **Draw**
- **Sound**

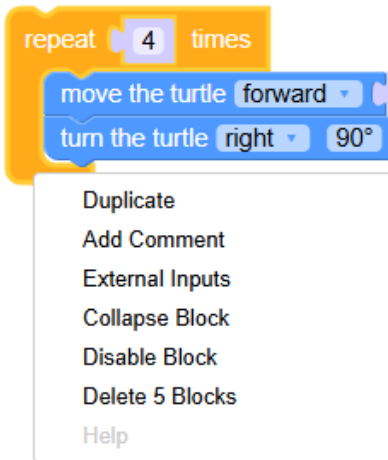
Let's learn some more block commands!

But first, here are some tips for power coders that will save you time and work.

Tips for Power Coders

Tip #1: If you select a block and press the Delete key, the block will go into the Trash can. You don't have to drag it there. You may find that short-cut helpful, but you can't undo it.

Tips #2: If you right-click a block, you will see some choices that will help you code more quickly and easily.

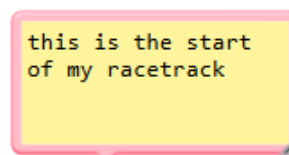


Right-click the border of the code, the golden border, in this case, for a useful context menu. You can do the same with an individual block as well. Because each block can vary, the options may be different from those shown here.

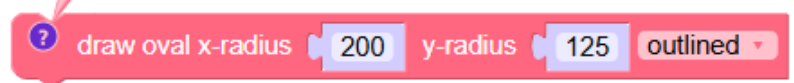
From this context menu, you can choose to:

- **duplicate** or make a copy of the block, a real time-saver when you want a similar block of code, which you can then edit to do a slightly different job
- **add a comment**, which can be useful to help other people understand your code (as well as a good reminder for yourself!)
- use **Inline Inputs** to put the inputs on one line; the opposite is **External Inputs** (see below)
- **collapse the block** so it takes just one line (you can expand it later); this can be very helpful with your code gets long
- **disable the block**, which keeps it around, but removes the Logo Code and won't run it; you can enable it with another right-click on the block
- **delete all the blocks** that are in the selection (you can't undo this one!)

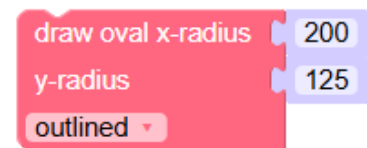
If you choose Add Comment, a question mark will appear in the upper left of the block. Click the question mark and type a note. It's a great way to document your code.



This shows the **Inline Inputs** option.



This shows the **External Inputs** option.



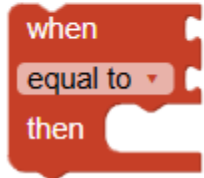
Let's get coding!

Events

These blocks allow you to wait for things to happen and then react to those situations. Handling events can seem rather complicated at first, but once you try some examples and edit them to do slightly different things, you will see how they work.



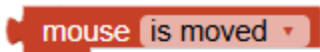
When blocks take an input about a situation that might change. You can tell Logo Blocks what to do what under certain circumstances.



The events you can look for are shown below.

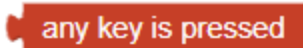


This event detects if the size of the graphics panel changes.

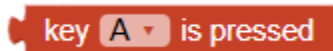


- ✓ is moved
- is dragged
- left button clicked

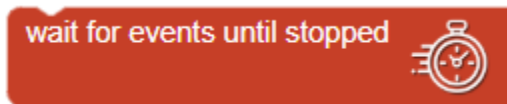
This event detects if the mouse is moved, dragged, or its left button is clicked. You can choose the condition.



This event is triggered if the user presses any key.



This event is triggered if the user presses a specific key. You can set the key to any letter (works with both upper and lower case), number, the space bar, or an arrow key.



This block causes the program to halt and wait for the specific event to occur.

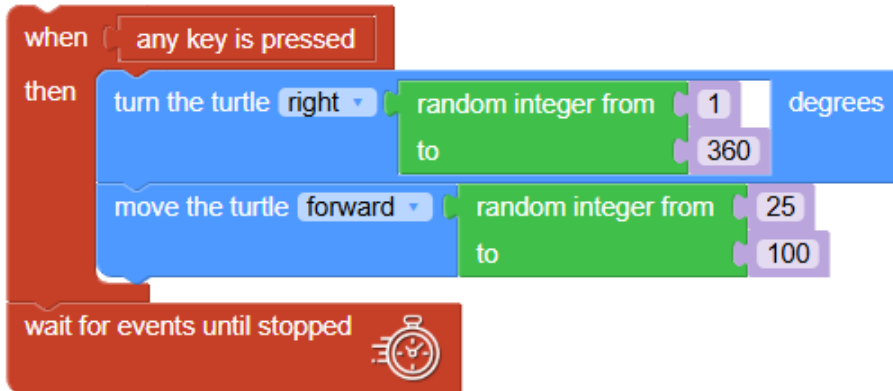
Always put this block at the bottom of your code when using events.

TIP: If you use a *when* block, don't put it inside a *forever* block. That will keep Logo busy, which you do not want.

Let's see some examples of these blocks in use. You can try these examples as is and then edit the blocks to act differently. It's up to you!

Example A:

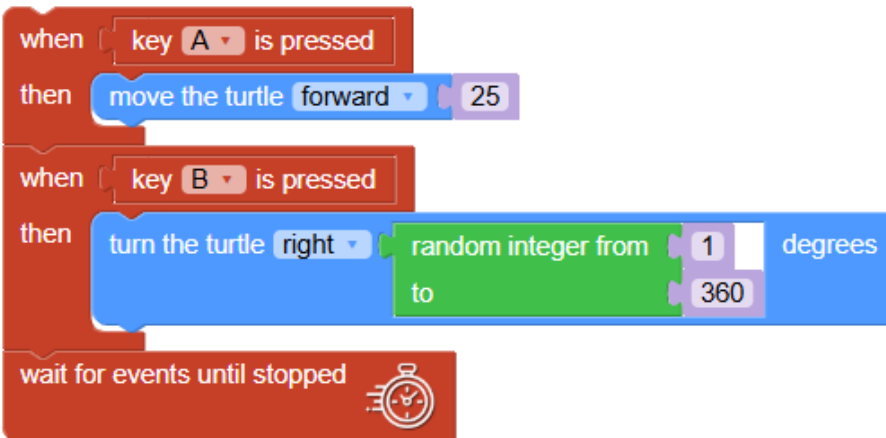
In this example, pressing any key moves and turns the turtle random amounts. It doesn't pay attention to what keys are pressed. Click the red **Stop** button to end the program.



Logo Code

```
1 WHEN [KEY] [  
2     RIGHT (RANDOM 1 360)  
3     FORWARD (RANDOM 25 100)  
4 ]  
5 WAIT -1
```

If you want different keys to perform different actions, use blocks like this:



Logo Code

```
1 WHEN [KEY = "A"] [  
2     FORWARD 25  
3 ]  
4 WHEN [KEY = "B"] [  
5     RIGHT (RANDOM 1 360)  
6 ]  
7 WAIT -1
```

Pressing the A key moves the turtle forward 25 steps. Pressing B turns it a random amount.

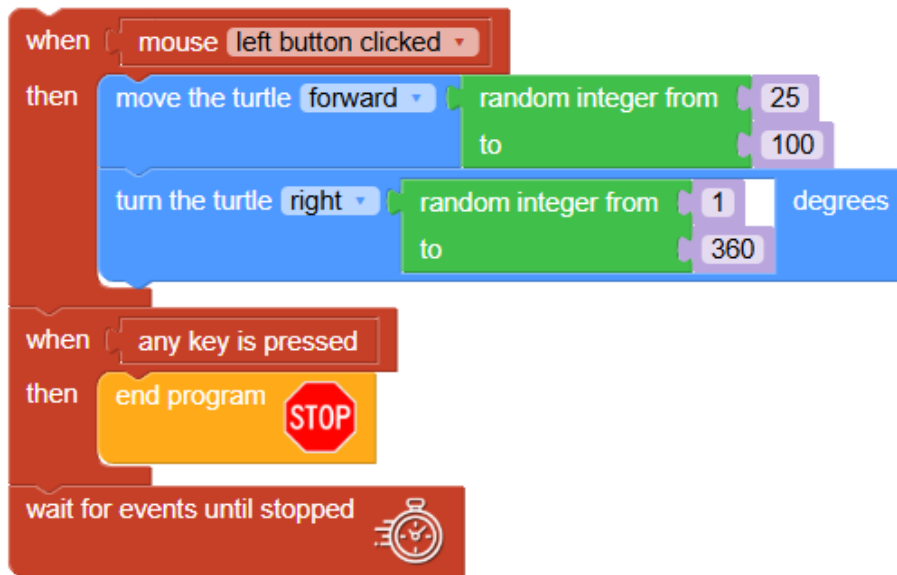
Explore different keys to cause different actions.

Example B:

In this example, clicking the left mouse button moves and turns the turtle in random amounts.

Pressing any key stops the program.

The speed is set to 0.9 so you watch the turtle draw.



```
Logo Code
1 WHEN [MOUSE CLICKED] [
2     FORWARD (RANDOM 25 100)
3     RIGHT (RANDOM 1 360)
4 ]
5 WHEN [KEY] [
6     TOPLEVEL
7 ]
8 WAIT -1
```

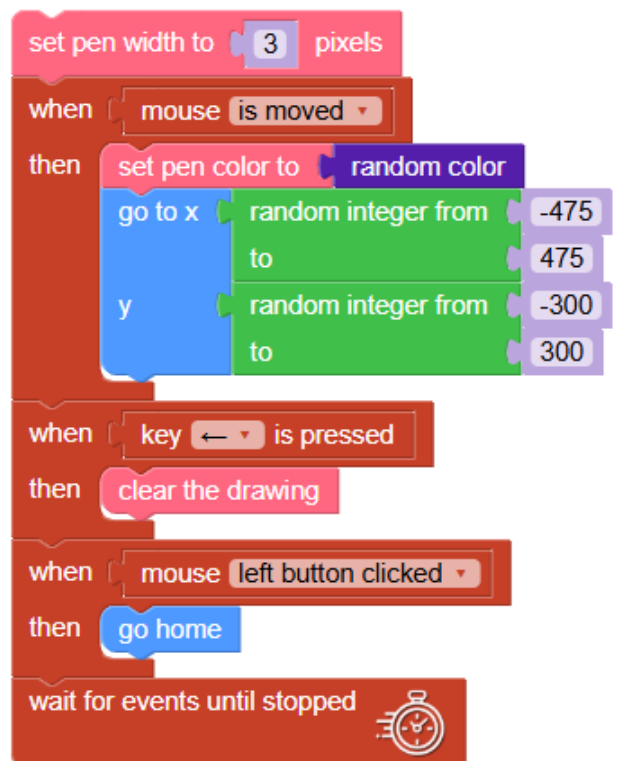
Example C:

In this example, moving the mouse draws a line to a random x-y coordinate location in a random color.

Pressing the left arrow key erases the drawing.

Clicking the mouse moves the turtle to its home in the middle of the screen.

You may wish to explore different turtle speeds.



```
Logo Code
1 SETWIDTH 3
2 WHEN [MOUSE MOVED] [
3     SETPC PICK COLORS
4     SETXY LIST (RANDOM -475 475) (RANDOM -300 300)
5 ]
6 WHEN [KEY = ARROWLEFT] [
7     CS
8 ]
9 WHEN [MOUSE CLICKED] [
10    HOME
11 ]
12 WAIT -1
```

Example D:

In this example, pressing A moves and turns the turtle and prints the letter A on the screen. Pressing B moves the turtle a different distance and prints the letter B on the screen.

Pressing the number 9 erases the screen.

```
Logo Code
1 SETFONT "TIMES 24 0
2 WHEN [KEY = A] [
3     TURTLETEXT "A
4     FORWARD 50
5     RIGHT 90
6 ]
7 WHEN [KEY = B] [
8     TURTLETEXT "B
9     FORWARD 100
10    RIGHT 45
11 ]
12 WHEN [KEY = 9] [
13     CS
14 ]
15 WAIT -1
```

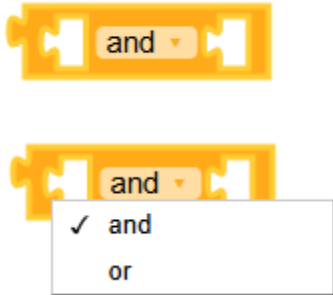
The image shows a Scratch-style block-based code editor. The code is as follows:

- set turtle font to Times, size 24 pixels
- when key A is pressed
 - then draw text " A "
 - move the turtle forward 50
 - turn the turtle right 90°
- when key B is pressed
 - then draw text " B "
 - move the turtle forward 100
 - turn the turtle right 45°
- when key 9 is pressed
 - then clear the drawing
- wait for events until stopped

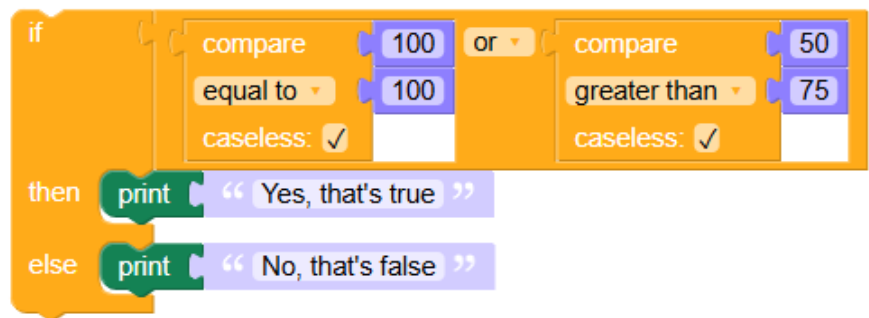
Flow

Use these blocks to control the direction the program goes. *If/Then* blocks are covered in the Intermediate Mode tutorial, as are *compare*, *repeat*, *forever*, *wait*, and *end program*.

Here are the new blocks introduced in the Advanced Mode.



This block, known as a *logical operator*, reports either True or False depending on the information you give it. You can compare two pieces of information and connect them with either *and* or *or*. Here is an example.

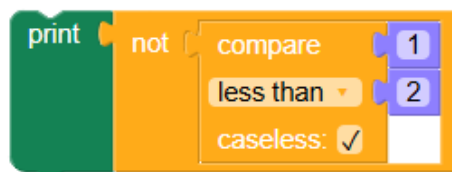


If you use *or*, you'll see "Yes, that's true" because only one of the two comparisons needs to be True for the block to report True.

If you change the *or* to *and*, you will see "No, that's false", because even though 100=100, 50 is not > 75. Both cases would have to be True for the combined result to be True.



This *not* block is another *logical operator*. It reports the opposite of the True/False information you give it. For example, in this case, it reports False because the comparison itself is True and the *not* that comes before the comparison reports the opposite of that comparison.



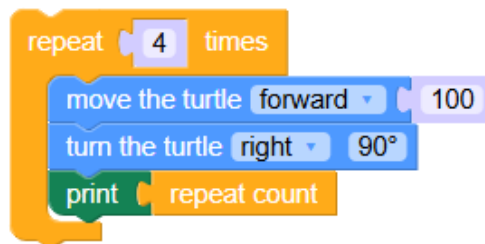
If you change *less than* to *greater than*, it will report True, the opposite of the result of the comparison.

If your head is beginning to hurt as you try to understand these logical blocks, don't worry! You may not need to use them often, but they can come in handy when you do!

repeat count

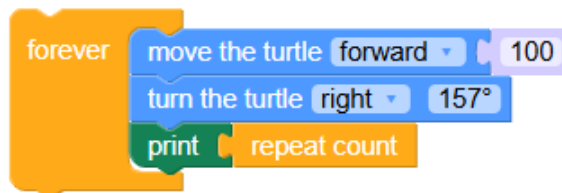
The *repeat count* block lets you keep track of the number of times the code is run through the loop. It counts the number of times the code runs a *repeat* loop or a *forever* command.

In this example, you will see the numbers 1 through 4 display in order, one for each time the repeat loop is run.



In the next example, a *forever* block is used.

Using *repeat count* with forever may help you figure out how many times you want to repeat a set of blocks.

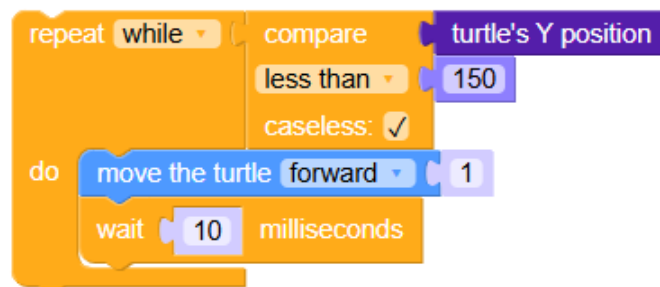


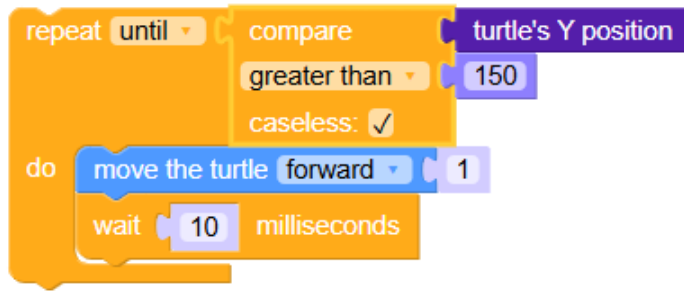
repeat while

do while
 until

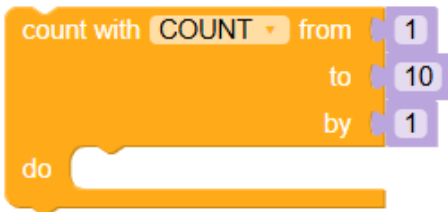
This block can be set to *while* or *until*. You can use it to repeat a set of commands until or while a condition is met.

For example, can you see how both these sets of code produce the same result? They are different in two spots. Look at the code carefully to find them.





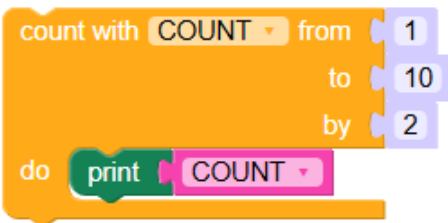
The *count* block uses a variable named COUNT to keep track of numbers. You can tell it to count by 1s, 2s, 3s, 10s, 100s, or any number. Tell it what number to start at and when to end.



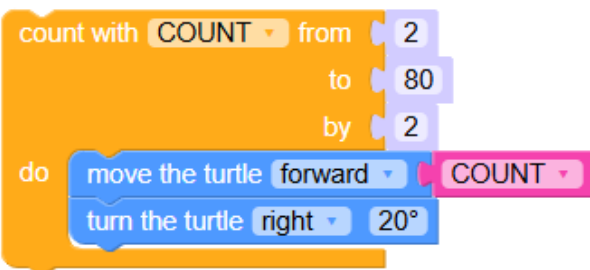
Here is a simple example to start with. It simply counts from 1 to 10.



The numbers appear one at a time in an alert box. If you change to a very large *to* number and don't want to finish the sequence, you can click the red **Stop** button in the alert box. Otherwise click **OK** to continue counting.

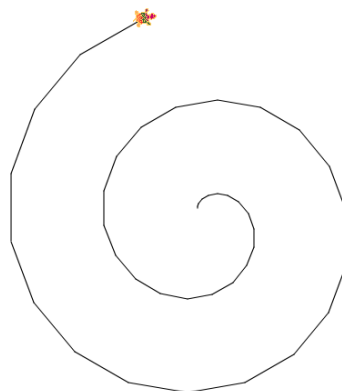


To try this example, you'll find the *print* block in the Output section. The **COUNT** variable is in the Variables section.



Next is a more complicated use of this block.

Can you imagine what the code will draw? Each step of the way, the turtle moves forward the value of COUNT, which increases by 2 every time. This is a type of loop. It uses Logo's FOR command.



```

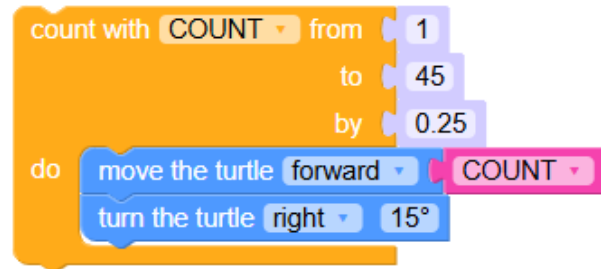
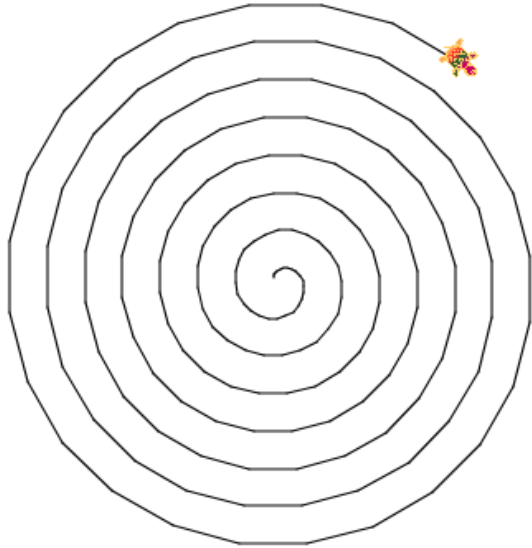
Logo Code
1 FOR [COUNT 2 80 2] [
2   FORWARD :COUNT
3   RIGHT 20
4 ]

```

Experiment with the numbers and create your own version!

Can you figure out how to create this variation?

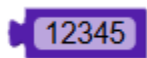
Notice that you can use a number less than 1 for the *by* value. This code uses .25. That means that the forward distance increases by a quarter of a step each time: 1, 1.25, 1.5, 1.75, 2, 2.25, 2.5, etc., all the way to 45.



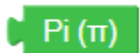
```
Logo Code
1 FOR [COUNT 1 45 0.25] [
2   FORWARD :COUNT
3   RIGHT 15
4 ]
```

Math

With these **Math** blocks, you can work with numbers in many different ways. Even though a couple of these blocks were covered in the [Intermediate tutorial](#), they are also included here.



Use this block when you want to insert a number into a block. You can change 12345 to any number you want. You have seen and likely used this block before.

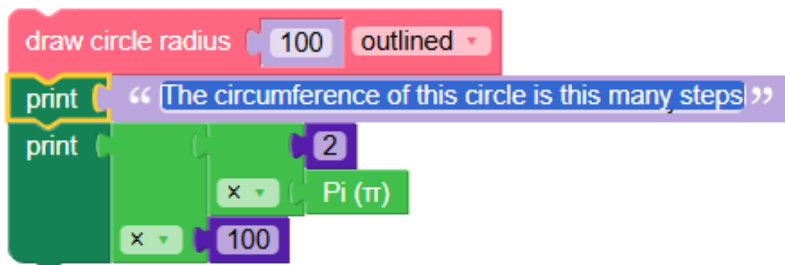


The block gives you the value of π , which is written as π . π is a value used to compute the circumference and area of circles.

The circumference of a circle is $2\pi R$, or 2 times π times the length of the radius.

The area of a circle is πR^2 , or π times the radius squared.

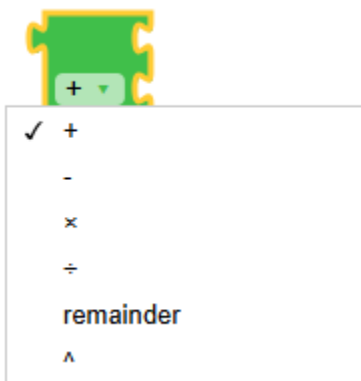
Here is code that computes the circumference of a circle.

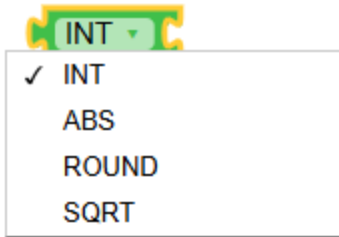


Can you write the code to figure out its area using π ?

Use the **operations** block when you want to do math. Connect a number block (see above) in both spots. You can then add, subtract, multiply, or divide them, find the remainder, or use an exponent. Three to the second power, or three squared, is an example of an exponent, written like this: 3^2 . It means 3×3 or 9.

For more examples, see the [Intermediate tutorial](#).





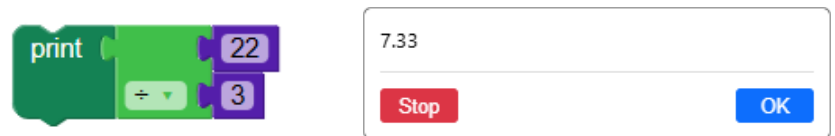
This multi-function block gives you many ways to format and manipulate numbers.

INT is short for integer. This block takes a number as input and returns a whole number, like this:

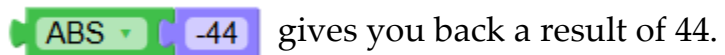


The input is 22/3. The integer result is 7.

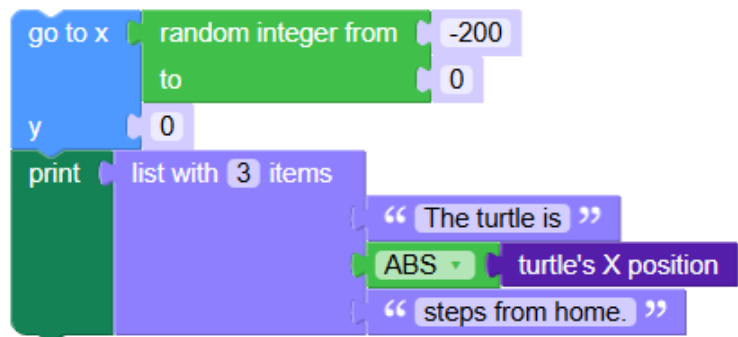
If you remove the *INT* block, the result is no longer a whole number. It is a decimal. See the difference?



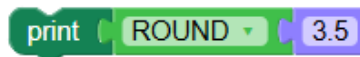
ABS means absolute value. The absolute value of a number is always a positive number.



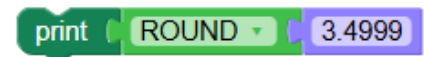
You might use this if you want to determine how far the turtle is from a target.



ROUND reports the nearest integer (whole number) to its input. If the decimal portion is greater than or equal to .5, the number rounds up. Otherwise, it rounds down.

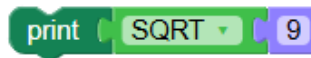


This reports 4.

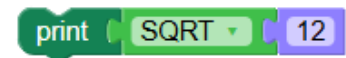


This reports 3.

SQRT reports the square root of the input number. A number squared is the number times itself. Finding a square root is the opposite of squaring a number. It finds what number times itself is the input.

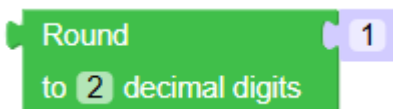


The square root of 9 is 3.

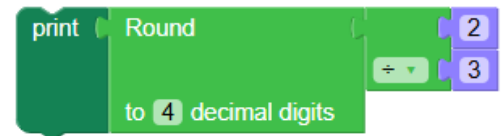
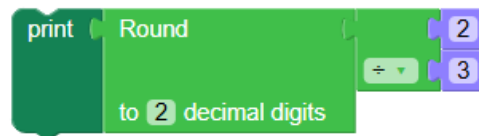


The square root of 12 is 3.46.

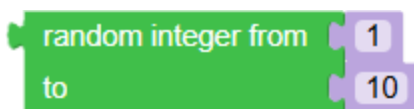
If the square root is a whole number, it is called a perfect square, such as 9 in the example above.



Use the *precision* block to return a number with the number of decimal places you want. Compare these:



You can use this block to report *Pi* to up to 15 places!



You've already seen examples of using the *random* block. Anytime you need a random number, just replace the current value with this block. Enter numbers for the range you want and see what happens!

Words and Lists

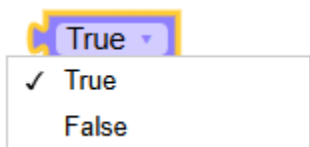
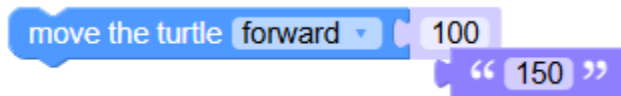
Inputs are pieces of information you use with blocks or procedures you write. The information can be numbers, letters, collections of words, colors, and much more, as you will see. You are already familiar with some of them.



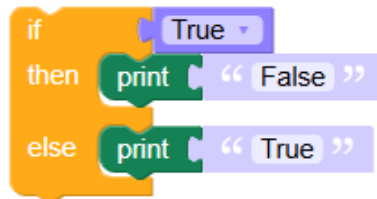
These two blocks are designed for numbers and letters. You can't enter letters into the block with 123. If you try, the background turns red, like this:



You can enter numbers into the text box, but you can't do math with them. They are no longer numbers. Logo Blocks won't let you connect a text box to a **forward** block, for example:



This block checks a condition and reports either True or False. Here's an example of using it.



About Words and Lists

Many of these blocks use **words** and **lists**. A **list** is a collection of words or numbers, or even other lists. There are spaces between the items in a list. A list can even contain another list. A **word** is a numbers or letters with no spaces between them, like feb26.

About Alert Boxes

You may notice that the Logo Code often includes this text when you use the *print* block,

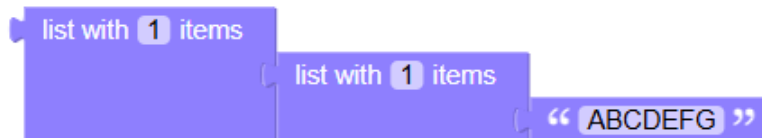


print displays its text in an alert box on the screen. (In the full Logo language, the PRINT command would put the text in the Listener window, which Logo Blocks doesn't have.) ALERT generally sends back information to Logo, so the IGNORE part of the Logo instruction means to ignore anything that the ALERT box might report back to it. You can ignore the `IGNORE ALERT` code.

Let's learn more blocks that use words and lists.

list with 0 items

Initially, the list block has no items. You can set the number of the items you want to include in the list. Here are some examples.



Logo Code

```
1 LIST `Yes` `No`
2
3 (LIST `dog` `cat` `bird`)
4
5 (LIST (LIST "ABCDEFG))
```

The instructions in the Logo Code shows that it places parentheses around a list with more than or fewer than two items, which is the standard number of inputs for a list. If the text is in upper and lower case, it is called a string and starts and ends with a backquote, like this: ``Yes``. If all the characters are uppercase or are letters, it starts with a quote mark.

word with 0 items

A word is a sequence of letters and/or numbers with no spaces between them. Here are some examples.

print word with 2 items

- “rain”
- “bow”

rainbow

Stop OK

print word with 3 items

- “1”
- “0”
- “1”

101

Stop OK

IGNORE ALERT WORD `rain` `bow`

IGNORE ALERT (WORD "1 "0 "1)

item count of

Item count of tells you how many items are in a word or list.

print item count of list with 3 items

- “chocolate”
- “vanilla”
- “strawberry”

3

Stop OK

print item count of word with 4 items

- “this”
- “is”
- “one”
- “word”

13

Stop OK

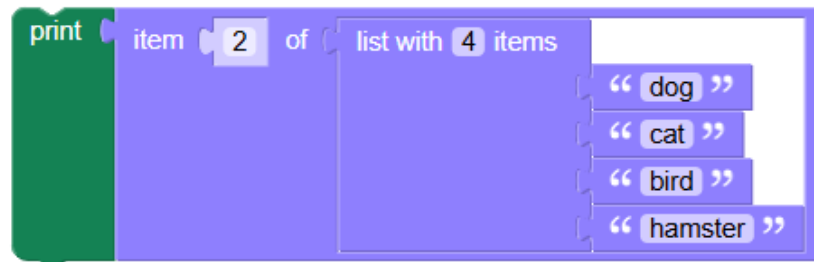
Are you surprised by that result? The result is 13 because it is counting all the letters in the word. If you print the word, like this:



You'll see why the result is 13.



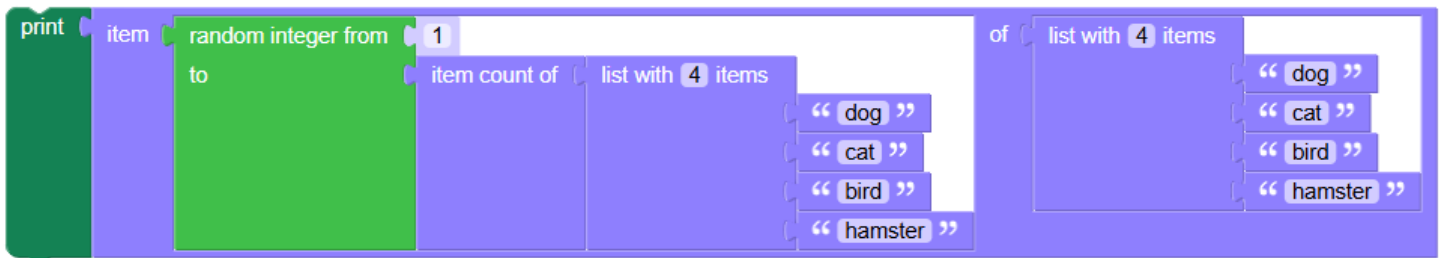
The *Item* block gives you the part of the list you specify.



IGNORE ALERT ITEM 2 (LIST `dog` `cat` `bird` `hamster`)



To choose a random item from a list, you could use this strategy:



But that is a bit more complicated than it needs to be! Let's try using a variable (much more about them later).

```

set ANIMALS to list with 4 items
  "dog"
  "cat"
  "bird"
  "hamster"
print item random integer from 1 to item count of ANIMALS

```

```

is empty

```

The block tells you if a word or list is empty.

```

print list with 0 items is empty

```

this reports True

```

print word with 1 items "goldfish" is empty

```

this reports False

```

first item of

```

The *first/last* block tells you the first or last item in the word or list you give it. You can use it with variables to save an item before removing it from a list (as in the next block).

```

print first item of list with 3 items
  "gold"
  "silver"
  "bronze"

```

The first item in this list is **gold**, which is the result you get. Change *first* to *last* and you'll get **bronze**.

It works with words as well as with lists. Here, you get back the first letter of the word, P.

```

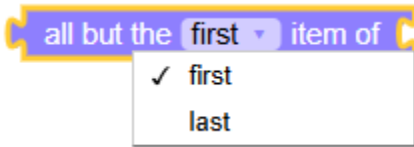
print first item of "Paddington"

```

```

P

```



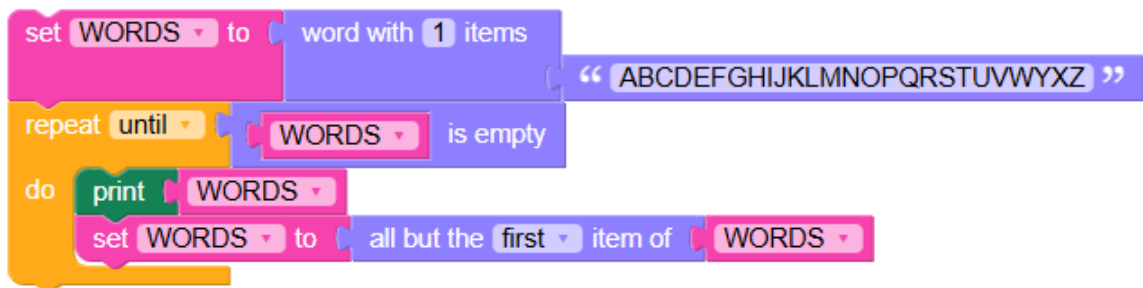
This block, which uses BUTFIRST and BUTLAST Logo commands is fun to use.



Can you think of an input that produces a different word when you use all but the *last* item of another word? Here's one. Think of a different one and try it.

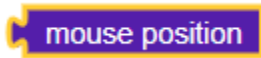


Here is a more complicated use of this block. It removes a letter from the beginning of the alphabet one at a time, repeatedly, until there are no more letters left. It uses variables, which you will meet later. See if you can try it now!

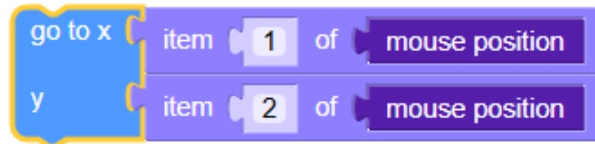


Input

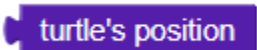
Inputs are pieces of information that Logo Blocks knows about. You use this information with blocks or procedures you write. The information can be numbers, letters, collections of words, colors, and much more, as you will see. You are already familiar with some of them.



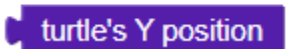
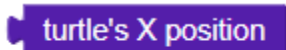
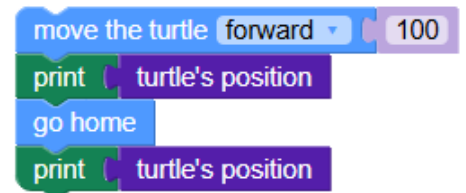
The *mouse position* block reports the position of the mouse. It is a list of two numbers: its x-coordinate and its y-coordinate. Here is an example of how you could use it.



Can you figure out how to use this code and the *forever* block to draw with the mouse?

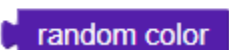
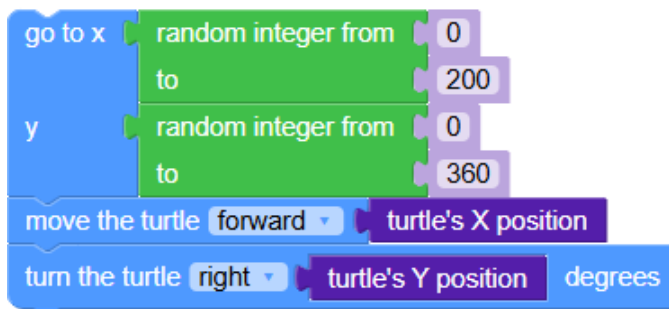


This is similar to the block that reports the position of the turtle: a list of two numbers representing its x- and y-coordinates.



Using these two blocks, you can work the turtle's x- and y-coordinates independently.

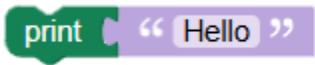
Here is a way to draw some random turtle art! Click the **Run** button many times to see the drawing, or put these blocks inside a *repeat* block.



You've already met these color blocks. They are included in the **Draw** category as well. Because they serve as inputs, they are included here, too.

Output

The Output section gives you a block that presents information. Output is the opposite of input, where you give the computer information. Now, you are telling it what to output.



The **print** block can accept either numbers or words. The text is displayed in an alert box.

Logo coding info:

If you type uppercase and lowercase letters in the **print** block, the text is kept that way. Logo treats it as a *string*.

If the text is all capital letters, Logo treats it as a *word*.

You can see the difference in the Logo Code.

The `backquotes` keep the text as you entered it.

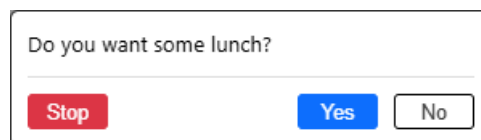
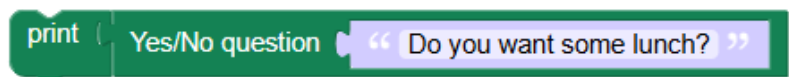
The "quote mark keeps the word in all uppercase.



You don't have to worry about this. It just how Logo works! If you go on to use the full Logo language, this will be useful information to remember.



This block lets you get a response to a Yes/No question. If the user presses Yes, it reports TRUE. If the user presses No, it reports FALSE.



You can make decisions and respond based on which button is clicked.

```

if Yes/No question " Do you want some lunch? "
then print " So am !! Let's get a sandwich. "
else print " OK. Maybe we can eat together soon. "

```

The *then* instructions run if Yes is clicked.
The *else* instructions run if No is clicked.

```

Display and enter a value as upper case: 
Text " "

```

This block displays text, such as instructions or a question, and waits for the user to type into a text field. What the user types can be printed, used in an *if/then* block, or as the input to a variable. Let's see some examples.

```

print Display and enter a value as upper case: 
Text " Type this word: apple "

```

If you check the "upper case" box, the text will be returned to you in uppercase characters (this doesn't apply to numbers or symbols).

```

if compare Display and enter a value as upper case: 
    equal to " Boston "
    caseless: 
then print " That's right! "
else print " Sorry, the capital of Massachusetts is Boston. "

```

```

set COLOR to Display and enter a value as upper case: 
    " What is your favorite color? "
print list with 3 items
    " I like "
    COLOR
    " too! "

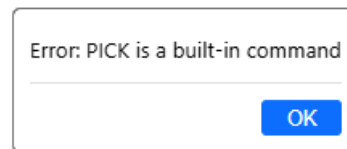
```

Perhaps some of these examples will inspire you to create interactive word games!

Procedures

A procedure is a set of code that has a name.

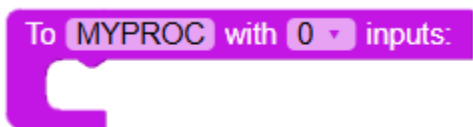
Note that you cannot name a procedure with the same name as a command in the Logo language. Of course, you don't know what those names are, but Logo Blocks will tell you if you can't use that name. If you enter a name with a space in it, Logo Blocks will replace the space with an underscore character.



To run the code in your procedure, you need to call it by its name. It is very efficient to create procedures, as you will see!

Logo Blocks lets you create procedures that have no inputs or the number of inputs you want.

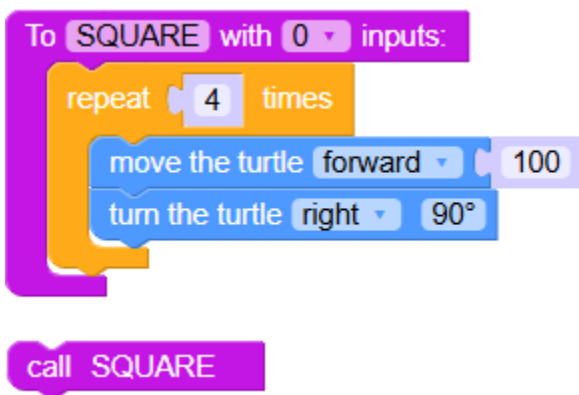
Here are some examples of how to create and use procedures, with and without inputs.



Let's create a simple procedure to draw a square. Drag this MYPROC block to the workspace area.

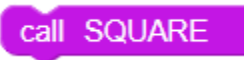
Change MYPROC to SQUARE so that it has a meaningful name. Keep the number of inputs at 0.

Add blocks to draw a square inside the procedure. You can use the *repeat* block, along with *forward* and *turn* blocks. Your procedure might look like this:



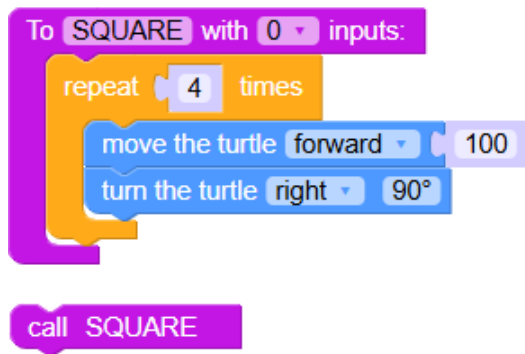
If you click **Run**, nothing happens. Your procedure is now defined, but you haven't told Logo Blocks to use it.

If you look at the Procedures section, you see a new block:

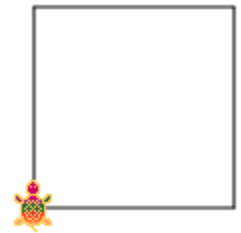


Use this block to tell the computer to run your SQUARE procedure.

Your workspace, Logo Code panel, and Graphics panel look like this:



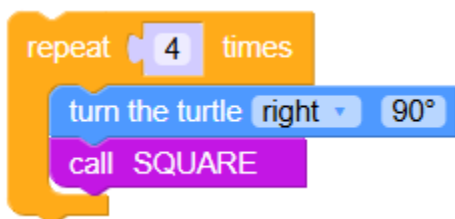
```
1 TO SQUARE
2   REPEAT 4 [
3     FORWARD 100
4     RIGHT 90
5   ]
6 END
7
8 SQUARE
```



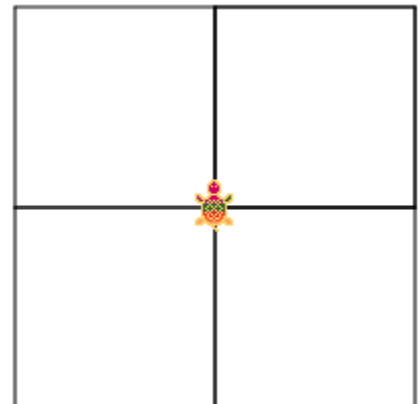
Congratulations! You have just created your first procedure!

How can you use your procedure? Just use its name, SQUARE, whenever you want a square. Don't delete the SQUARE procedure. You need to keep it around to call it again. If you delete it, Logo Blocks will forget that.

Here is an example of using a procedure inside a *repeat* block.

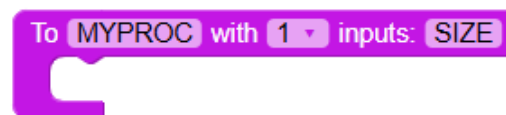
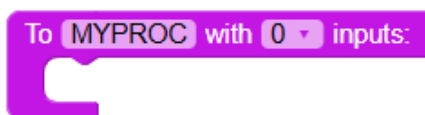


```
Logo Code
1 TO SQUARE
2   REPEAT 4 [
3     FORWARD 100
4     RIGHT 90
5   ]
6 END
7
8 REPEAT 4 [
9   RIGHT 90
10  SQUARE
11 ]
```



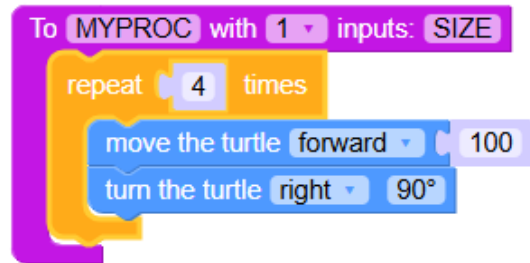
What if you want to have a procedure that will draw any size square you want? To do that, you need to use a *variable*. A *variable* is a placeholder. You give it a name, like SIZE.

Start with the MYPROC procedure. Edit the procedure to use 1 input named SIZE.



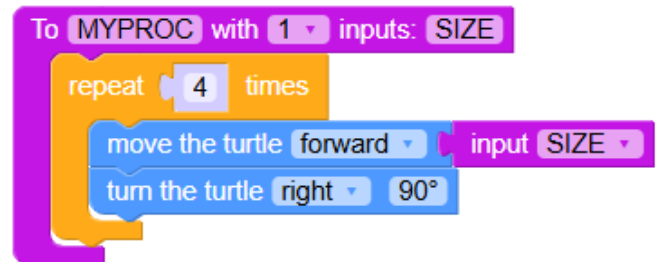
You are creating a procedure that uses a variable. Think of a variable as a container that can hold information, such as a number, word, or list. A variable that is part of the name of a procedure is called a *local variable* because only this procedure can use it. Later you will learn how to create *global variables*, which can be seen and used by any procedure in your program.

Next, add the blocks to draw a square. Your procedure will now look like this:



Now we need to tell the procedure to use the variable. In the **Procedures** section is a block for an input.

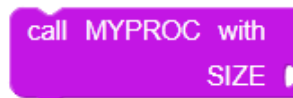
Drag the *input* block to replace the 100 in the *forward* block. It now looks like this:



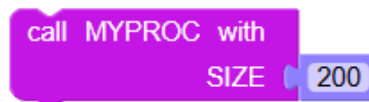
Logo Blocks even knew to call it SIZE!

Now, to use this new procedure that has a variable, you need to call the procedure with a number. This gives a value to the variable named SIZE. Your code will now use the number you give the procedure whenever it sees the name SIZE.

In the **Procedures** area, look for this block that you can use to call your procedure.



Go to the **Math** section and grab the number block. Drag it next to the variable name SIZE. Change the number to the size you want. For example, try 200.

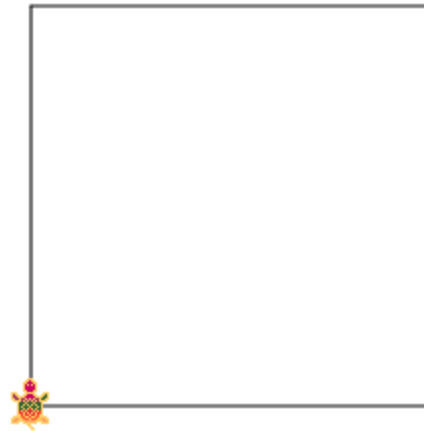


Now click **Run**. You will see a size 200 square. Also look at the Logo Code to see how Logo created the commands for the procedure. It puts "dots," or a colon, before a variable name.

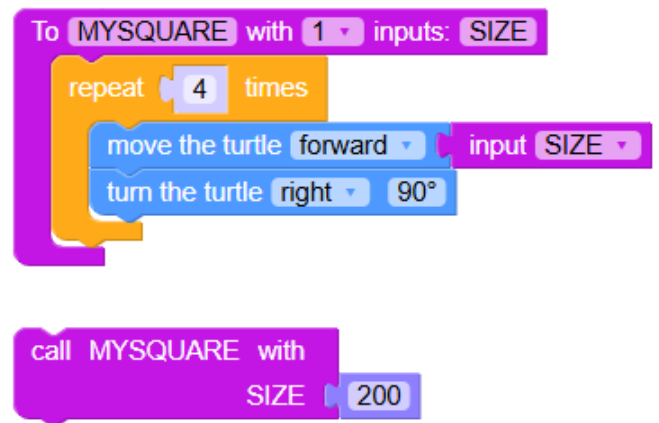
```

Logo Code
1 TO MYPROC :SIZE
2   REPEAT 4 [
3     FORWARD :SIZE
4     RIGHT 90
5   ]
6 END
7
8 MYPROC 200

```

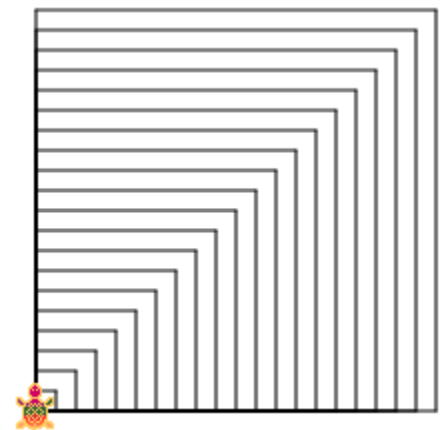


You can rename the procedure at any time. Just click in the MYPROC text field and type the new name. We'll rename it to MYSQUARE. The procedure, its calling block, and the Logo Code will change to match the new name.



Try calling the MYSQUARE procedure with different input numbers.

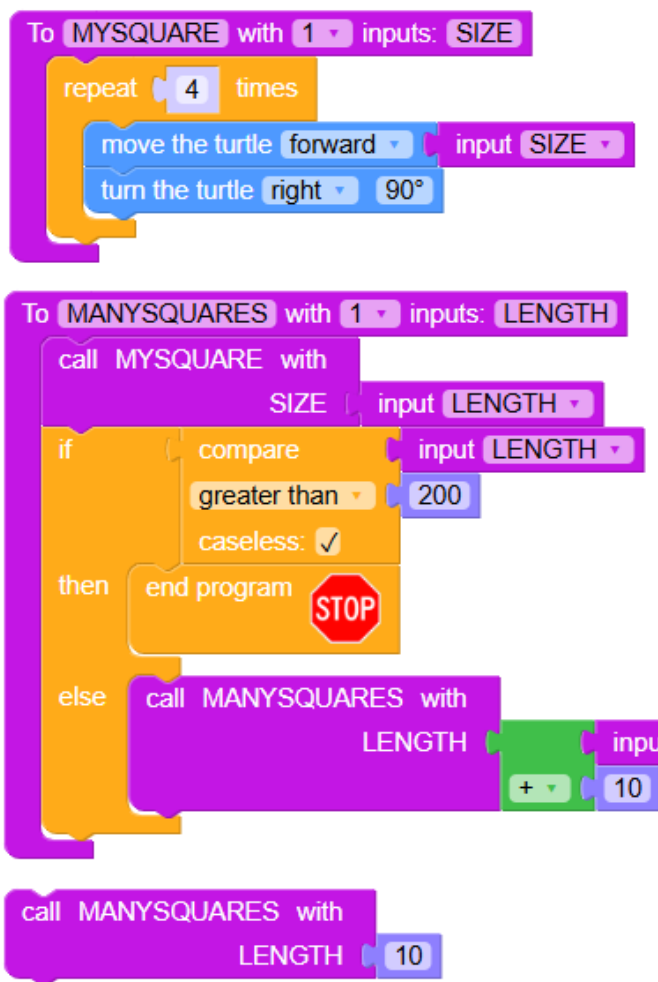
This design was created using inputs of 10, 20, 30, 40, etc., all the way to 200!



Let's save you a lot of typing by automating the process. Try the set of code below.

You'll need to create a new procedure, MYSQUARE, which takes an input for the size.

You also need to create a new variable that calls MYSQUARE multiple times with the length of the side growing by 10 each time. The program stops when the number is greater than 200.



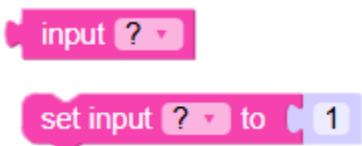
```

Logo Code
1 TO MYSQUARE :SIZE
2   REPEAT 4 [
3     FORWARD :SIZE
4     RIGHT 90
5   ]
6 END
7
8 TO MANY SQUARES :LENGTH
9   MYSQUARE :LENGTH
10  IF (.COMPARE :LENGTH 200 "GT TRUE) [
11    TOPLEVEL
12  ] [
13    MANY SQUARES :LENGTH + 10
14  ]
15 END
16
17 MANY SQUARES 10

```

Be sure to put the MYSQUARE procedure above the MANY SQUARES procedure.

Do you see how this code works? The Logo command TOPLEVEL in the Logo Code is the same as the *end the program* block. Play with the numbers to draw different shapes in different sizes.



These two blocks are colored pink, not purple, because they are related to variables, and the Variables section's color is purple. In the **Procedures** category, the variables you use are *local variables*, which only can be seen by the procedures you create with an input. (Next, in the Variables section, you will meet *global variables*, which are visible to your entire program.)

Here is a short program that uses these blocks.

```

To MYPROC with 1 inputs: PLURAL
  set input PLURAL to word with 2 items
  input PLURAL
  " s "
  print input PLURAL

call MYPROC with
  PLURAL " duck "

```

You call the procedure with a singular noun and it replaces that input with the word followed by the letter 's' to make it plural.

Try different singular nouns. Does this program create the correct plural for all of them? If not, perhaps you can code a program that handles exceptions to the rule!

```

if return

```

This program looks at a condition and returns a value that you define. This program uses it. An explanation follows.

```

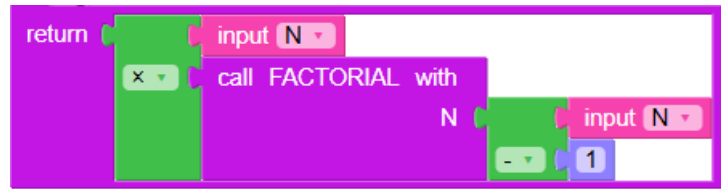
To FACTORIAL with 1 inputs: N
  if
    compare
      input N
      less than or equal to 1
      caseless: ✓
    return input N
  return
    input N
    ×
    call FACTORIAL with
      N
      -
      1
  print call FACTORIAL with
    N 5

```

The program is called FACTORIAL because it gives you the factorial of the number you call it with. The factorial of 5 is 120. A factorial is computed by multiplying the number by the number minus one repeatedly until the number reaches 1.

So, $5 \times 4 \times 3 \times 2 \times 1 = 120$.

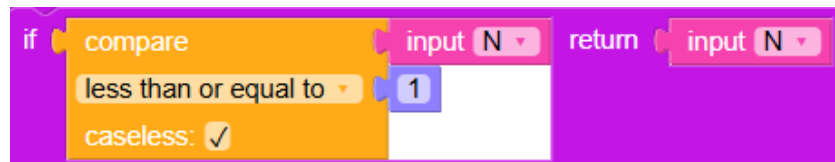
This program is recursive, which means that the definition of the program includes a call to itself. Do you see the call to FACTORIAL in the return section, here?



The Loo Code looks like this:

```
Logo Code
1 TO FACTORIAL :N
2   IF (.COMPARE :N 1 "LE TRUE) [
3     OUTPUT :N
4   ]
5   OUTPUT :N * (FACTORIAL :N - 1)
6 END
7
8
9 IGNORE ALERT FACTORIAL 5
```

You give FACTORIAL a starting number when you call it. The procedure calls this input N. The IF part compares the input N to the number 1.



If it is less than or equal to 1, where "LE means <= (Logo uses: .COMPARE :N 1 "LE TRUE.), the program returns the number stored by the variable N. If the number is still greater than 1, then FACTORIAL is called again, this time with the value of N minus 1.

Recursion can be complicated to understand at first. Adding a *print* block to follow the value of N might help.

The image shows a Scratch code block titled "To FACTORIAL with 1 inputs: N". It contains the following blocks:

- A "print" block with "input N" as the message.
- An "if" block with a "compare" block inside. The "compare" block has "less than or equal to" selected and "1" as the second input. The "if" block has a "return" block with "input N" as the message.
- A "return" block with a "x" block, a "call FACTORIAL with" block (with "N" as the input), and a "-" block with "1" as the input.

Below this is another "print" block with "call FACTORIAL with" and "N" as the message, and "5" as the output.

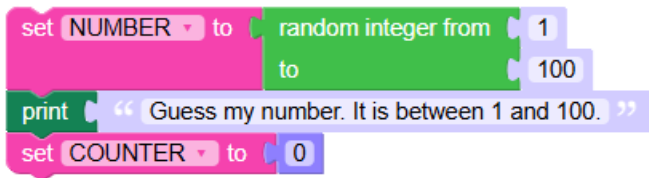


This final block in Procedures allows you to ignore any value you give it. You never know when you might need it!

Before we move to the final section called **Variables**, let's look at a more advanced version of the number guessing game introduced in the Intermediate tutorial.

Guessing Game

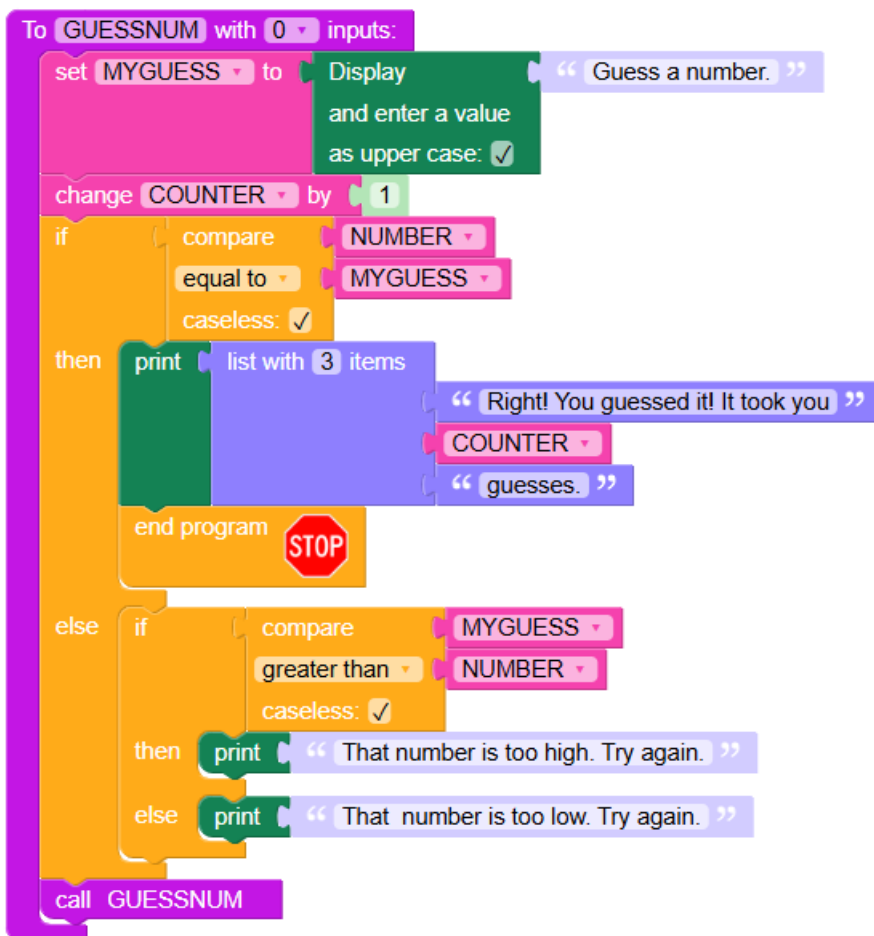
Do you remember the simple number guessing game we explored in the Intermediate tutorial? Here is a more advanced version. Let's take a look at it.



```
set NUMBER to random integer from 1 to 100
print "Guess my number. It is between 1 and 100."
set COUNTER to 0
```

First, we need to set up the program. We create a variable called NUMBER, which stores a random number from 1 to 100.

The *print* block gives the user instructions. We then set the COUNTER variable to 0. It will keep track of how many tries it takes to guess the number.



```
To GUESSNUM with 0 inputs:
  set MYGUESS to Display and enter a value as upper case:
  change COUNTER by 1
  if compare NUMBER equal to MYGUESS caseless:
    then print list with 3 items: "Right! You guessed it! It took you", COUNTER, "guesses."
    end program
  else if compare MYGUESS greater than NUMBER caseless:
    then print "That number is too high. Try again."
    else print "That number is too low. Try again."
  call GUESSNUM
```

Next, we create the main procedure, called GUESSNUM. The *display* block gives instructions and the player types a number. The variable MYGUESS stores that number.

We increase the variable COUNTER by one. It started at zero, so if the player guesses the correct number on the first try, the value of COUNTER is 1.

We compare the variable NUMBER (the answer) to the user's guess (MYGUESS).

If they are the same, we tell the user they got it right and tell them how many tries it took (the value of COUNTER). Then we stop the program.

If they get it wrong, we compare their guess (MYGUESS) to the answer (NUMBER). If the MYGUESS was greater than NUMBER, we tell the player and ask them to try again. If MYGUESS is too low, we tell them, and ask them to try again.

After we give the player the message, we call the GUESSNUM procedure again so they can try a different number. When a procedure calls itself, it is known as **recursion**.

call GUESSNUM

The last block in the program is to call GUESSNUM. That starts the program off!

Let's see what the Logo Code looks like.

```
Logo Code
1 MAKE "NUMBER (RANDOM 1 100)
2 IGNORE ALERT `Guess my number. It is between 1 and 100.`
3 MAKE "COUNTER 0
4
5 TO GUESSNUM
6 MAKE "MYGUESS READPROMPT `Guess a number.`
7 MAKE "COUNTER :COUNTER + 1
8 IF (.COMPARE :NUMBER :MYGUESS "EQ TRUE) [
9   IGNORE ALERT (LIST `Right! You guessed it! It took you` :COUNTER `guesses.`)
10  TOPLEVEL
11 ] [
12   IF (.COMPARE :MYGUESS :NUMBER "GT TRUE) [
13     IGNORE ALERT `That number is too high. Try again.`
14   ] [
15     IGNORE ALERT `That number is too low. Try again.`
16   ]
17 ]
18 GUESSNUM
19 END
20
21 GUESSNUM
```

MAKE is the Logo command to create or change a variable.

MAKE "NUMBER is set to the random number, which Logo gets using (RANDOM 1 100).

MAKE "COUNTER sets the value of the variable to 0. That is called initializing the variable.

When we increase the value of COUNTER, we use MAKE "COUNTER :COUNTER + 1.

When we refer to a variable that already has a value, we use "dots" or the : before its name.

The IGNORE before the ALERT command tells Logo it doesn't have to do anything with what is displayed. ALERT can give information back to Logo, but here, Logo can just ignore it.

IF needs something to look at; in this case it compares the guess to the random number. If the result is TRUE (they are the same), then Logo runs the instructions in the list that follows it. If the result is FALSE (the guess is wrong), then it runs the list after that. That list is comparing the guess to the answer so it can tell the player if the guess is too high or too low.

Logo is using this syntax or coding strategy to handle these conditional IF/THEN statements.

IF *this-condition-is-true* [then run this code] [else run this code]

TOPLEVEL is the same as Stop the program. That happens if the number is guessed!

If the guess is wrong, the procedure GUESSNUM is called to give the player another chance to guess the number.

Spend some time looking at the program so you can figure out how it works.

Variables

This category starts with just one button.



But that one button will open up a world of possibilities!

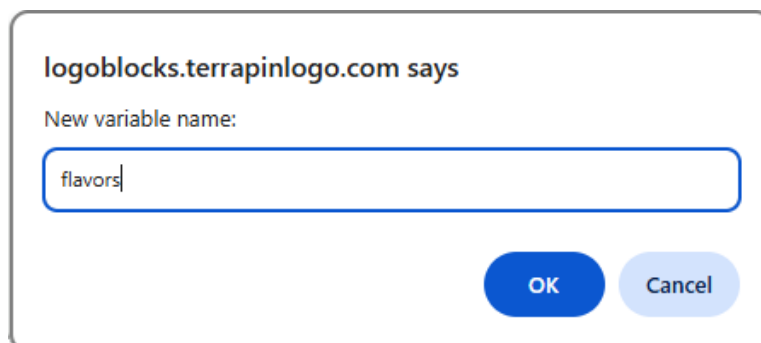
The variables you create with this button are called *global variables*. They can be seen and used by all the procedures in your project. That makes them very powerful.

To show you how to create and use variables, we're going to open an ice cream shop. The program will do the following:

- Ask the customer if they want an ice cream cone and respond appropriately to their answer (either Yes or No).
- Tell the customer they can choose two ice cream flavors.
- Have them choose the first flavor and then a second flavor.
- Tell them what they ordered.
- Thank them and tell them the cost.

Let's learn how to create the code.

You will need several variables to store the information. First, set up a list of flavors. Click the **Create variable...** button and enter **flavors** as its name.

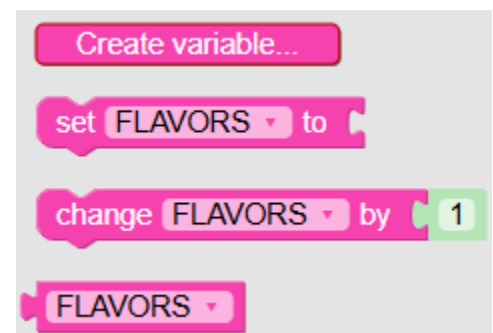


Look at the new blocks in the Variables section:

You can give FLAVORS a value using the *set* block.

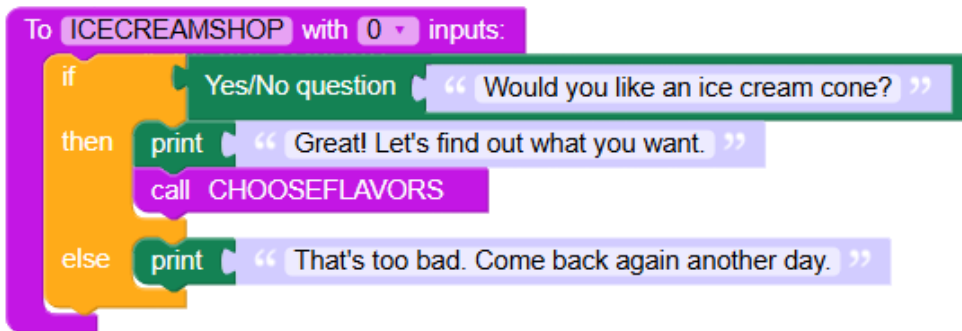
If FLAVORS were a number, you could make it bigger.

You can use the variable FLAVORS using the last block.



Create a procedure that starts the program. We'll call it ICECREAMSHOP.

It asks a person if they want an ice cream cone. If they do, we start the order process by calling a different procedure, CHOOSEFLAVORS. If they don't, the program ends because there is nothing else for it to do. You don't need to use the *end program* block.



```
To ICECREAMSHOP with 0 inputs:  
if Yes/No question "Would you like an ice cream cone?"  
then print "Great! Let's find out what you want."  
    call CHOOSEFLAVORS  
else print "That's too bad. Come back again another day."
```

Next, use the *set* block to make a list of flavors, like this (choose any flavors you want!):



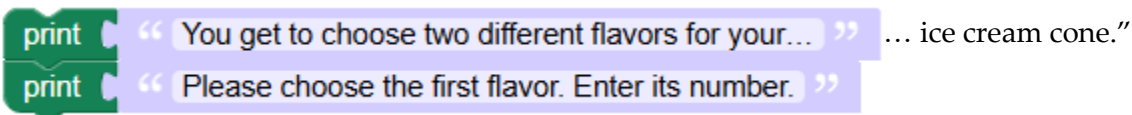
```
set FLAVORS to list with 5 items  
"1-chocolate"  
"2-vanilla"  
"3-strawberry"  
"4-rocky road"  
"5-mint chocolate chip"
```

Each one starts with a number so people don't have to type the words.

Next, we need a CHOOSEFLAVORS procedure. It will ask the customer to choose two flavors. We'll use new variables to store what they enter.

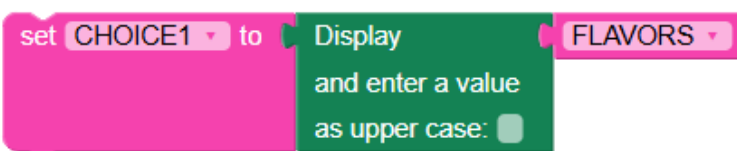
Use the *Create variable...* button to make variables called CHOICE1 and CHOICE2.

First, tell the customer what to do:



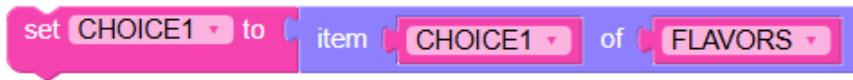
```
print "You get to choose two different flavors for your... " ... ice cream cone."  
print "Please choose the first flavor. Enter its number."
```

This block displays the list of flavors, stored in the FLAVORS variable, and stores their answer in variable named CHOICE 1.



```
set CHOICE1 to Display and enter a value as upper case: FLAVORS
```

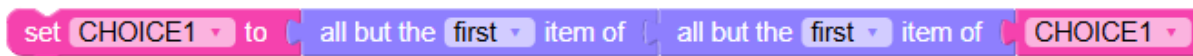
So, when the customer types a number to choose flavor, that number is stored in the CHOICE1 variable. But we need to convert the number to the name of the flavor and not its number. To do this use the *item* block, like this:



If they type 2, the value of CHOICE1 changes to become item 2 of FLAVORS. CHOICE1 goes from being 2 to being `2-vanilla`, the second item of the list of FLAVORS.

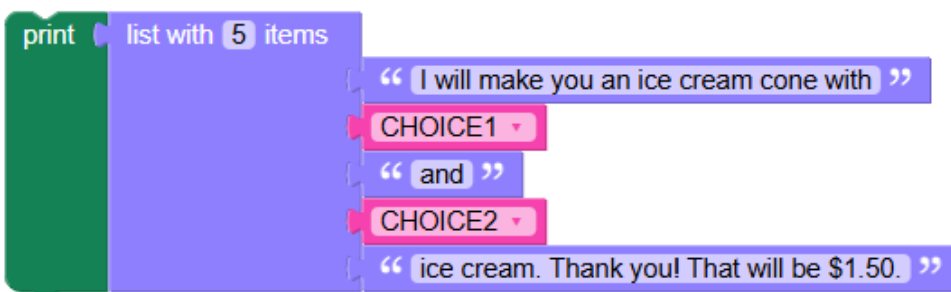
But how do we get rid of the number and hyphen before the name of the flavor? Do you remember the *all but the first item of* block? That is just what we need! But we have to use it two times. The first time removes the number, and we're left with `-vanilla`. There is still a hyphen before vanilla, so the second use of the *all but the first item of* block removes that.

So, this code turns the value of CHOICE1 from `2-vanilla` into just plain `vanilla`.



Now, we can use the same process to collect the second flavor choice. Use the same code, but use the CHOICE2 variable you created instead of CHOICE1. That lets us keep the two flavors separate, each with its own variable name.

Finally, tell them what you will make for them and how much it will cost.



Don't forget to call the main procedure to start the program!

Put this block last in your code. A purple 'call' block containing 'ICECREAMSHOP'.

Let's look at the complete program now (on the next page).

```

set FLAVORS to list with 5 items
  " 1-chocolate "
  " 2-vanilla "
  " 3-strawberry "
  " 4-rocky road "
  " 5-mint chocolate chip "

```

```

To CHOOSEFLAVORS with 0 inputs:
  print " You get to choose two different flavors for your... " ice cream cone.
  print " Please choose the first flavor. Enter its number. "
  set CHOICE1 to Display FLAVORS
    and enter a value
    as upper case:
  set CHOICE1 to item CHOICE1 of FLAVORS
  set CHOICE1 to all but the first item of all but the first item of CHOICE1
  print " Great! Now choose your second flavor. "
  set CHOICE2 to Display FLAVORS
    and enter a value
    as upper case:
  set CHOICE2 to item CHOICE2 of FLAVORS
  set CHOICE2 to all but the first item of all but the first item of CHOICE2
  print list with 5 items
    " I will make you an ice cream cone with "
    CHOICE1
    " and "
    CHOICE2
    " ice cream. Thank you! That will be $1.50. "

```

```

To ICECREAMSHOP with 0 inputs:
  if Yes/No question " Would you like an ice cream cone? "
  then print " Great! Let's find out what you want. "
    call CHOOSEFLAVORS
  else print " That's too bad. Come back again another day. "

```

```

call ICECREAMSHOP

```

The ICECREAMSHOP procedure has to come last because it refers to the CHOOSEFLAVORS procedure, which has to be defined or stored as a procedure before it can be used.

The Logo Code looks like this:

```
MAKE "FLAVORS (LIST `1-chocolate` `2-vanilla` `3-strawberry` `4-rocky
road` `5-mint chocolate chip`)

TO CHOOSEFLAVORS
  IGNORE ALERT `You get to choose two different flavors for your ice
cream cone.`
  IGNORE ALERT `Please choose the first flavor. Enter its number.`
  MAKE "CHOICE1 READPROMPT :FLAVORS
  MAKE "CHOICE1 ITEM :CHOICE1 :FLAVORS
  MAKE "CHOICE1 BUTFIRST BUTFIRST :CHOICE1
  IGNORE ALERT `Great! Now choose your second flavor.`
  MAKE "CHOICE2 READPROMPT :FLAVORS
  MAKE "CHOICE2 ITEM :CHOICE2 :FLAVORS
  MAKE "CHOICE2 BUTFIRST BUTFIRST :CHOICE2
  IGNORE ALERT (LIST `I will make you an ice cream cone with` :CHOICE1
`and` :CHOICE2 `ice cream. Thank you! That will be $1.50.`)
END

TO ICECREAMSHOP
  IF CONFIRM `Would you like an ice cream cone?` [
    IGNORE ALERT `Great! Let's find out what you want.`
    CHOOSEFLAVORS
  ] [
    IGNORE ALERT `That's too bad. Come back again another day.`
  ]
END

ICECREAMSHOP
```

Modify the program!

What could you add to this program? Try one or more of these ideas!

- Ask the customer if they want the ice cream in a cone or a cup.
- Allow the customer to choose 3 flavors.
- Add more flavors to the list.
- Change the program to offer just one-scoop ice cream cones.

Remember to save your program frequently!

Now you can create any project you can imagine! And send us your code for our [Gallery](#).