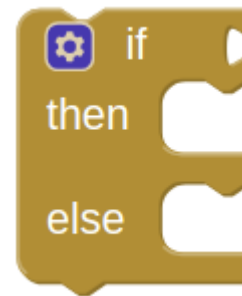




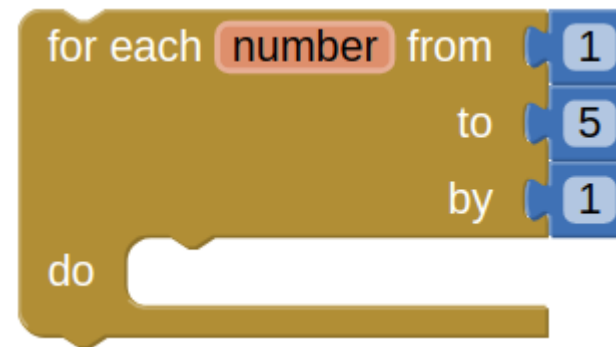
Tests a given condition. If the Condition is true, performs the actions in a given sequence of blocks; otherwise, the blocks are ignored.



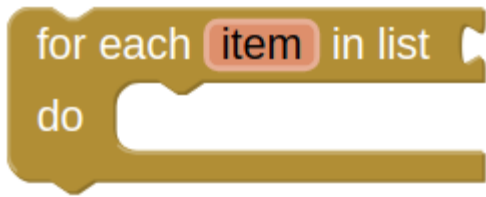
Tests a given condition. If the result is true, performs the actions in the -then sequence of blocks; otherwise, performs the actions in The -else sequence of blocks.



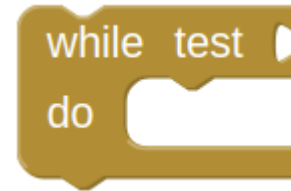
Tests a given condition. If the result is true, performs the actions in the -then sequence of blocks; otherwise tests the statement in the -else if section. If the result is true, performs the actions in the -then sequence of blocks; otherwise, performs the actions in the -else sequence of blocks.



Runs the blocks in the do section for each numeric value in the range starting at from and ending at to, incrementing number by the value of by each time. Use the given variable name, number to refer to the current value. You can change the name number to something else if you wish.



Runs the blocks in the do section for each item in the list. Use the given variable name, item, to refer to the current list item. You can change the name item to something else if you wish.



Tests the -test condition. If true, performs the action given in -do , then tests again. When test is false, the block ends and the action given in -do is no longer performed.



Tests a given condition. If the statement is true, performs the actions in the then-return sequence of blocks and returns the then-return value; otherwise, performs the actions in the else-return sequence of blocks and returns the else-return value.



Sometimes in a procedure or another block of code, you may need to do something and return something, but for various reasons you may choose to use this block instead of creating a new procedure.

evaluate but ignore result

Provides a "dummy socket" for fitting a block that has a plug on its left into a place where there is no socket, such as one of the sequence of blocks in the do part of a procedure or an if block. The block you fit in will be run, but its returned result will be ignored. This can be useful if you define a procedure that returns a result, but want to call it in a context that does not accept a result.

open another screen screenName

Opens the screen with the provided name.

open another screen with start value screenName
startValue

Opens another screen and passes a value to it.

get start value

Returns the start value given to the current screen.

This value is given from using
"open another screen with start value"
or
"close screen with value".

close screen

Closes the current screen.

close screen with value result

Closes the current screen and returns a value to the screen that opened this one.

close application

Closes the application.

get plain start text

Returns the plain text that was passed to this screen when it was started by another app. If no value was passed, it returns the empty text. For multiple screen apps, use get start value rather than get plain start text.

close screen with plain text text

Closes the current screen and passes text to the app that opened this one. This command is for returning text to non-App Inventor activities, not to App Inventor screens. For App Inventor Screens, as in multiple screen apps, use “Close Screen with Value”, not “Close Screen with Plain Text”.

true

Represents the constant value true. Use it for setting boolean property values of components, or as the value of a variable that represents a condition.

false

Represents the constant value false. Use it for setting boolean property values of components, or as the value of a variable that represents a condition.

not

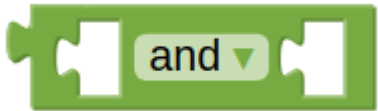
Performs logical negation, returning false if the input is true, and true if the input is false.



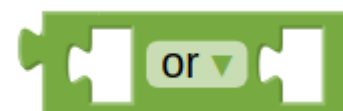
Tests whether its arguments are equal.



Tests to see whether two arguments are not equal.



Tests whether all of a set of logical conditions are true. The result is true if and only if all the tested conditions are true. When you plug a condition into the test socket, another socket appears so you can add another condition. The conditions are tested left to right, and the testing stops as soon as one of the conditions is false. If there are no conditions to test, then the result is true.



Tests whether any of a set of logical conditions are true. The result is true if one or more of the tested conditions are true. When you plug a condition into the test socket, another socket appears so you can add another condition. The conditions are tested left to right, and the testing stops as soon as one of the conditions is true. If there are no conditions to test, then the result is false.



Can be used as any positive or negative number (decimals included). Double clicking on the "0" in the block will allow you to change the number.



Tests whether two numbers are equal and returns true or false.



Tests whether two numbers are not equal and returns true or false.



Tests whether the first number is greater than the second number and returns true or false.



Tests whether the first number is greater than or equal to the second number and returns true or false.



Tests whether the first number is less than the second number and returns true or false.



Tests whether the first number is less than or equal to the second number and returns true or false.



Returns the result of adding any amount of blocks that have a number value together. Blocks with a number value include the basic number block, length of list or text, variables with a number value, etc. This block is a mutator and can be expanded to allow more numbers in the sum.



Returns the result of subtracting the second number from the first.



Returns the result of multiplying any amount of blocks that have a number value together. It is a mutator block and can be expanded to allow more numbers in the product.



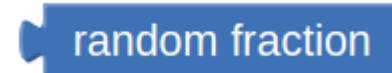
Returns the result of dividing the first number by the second.



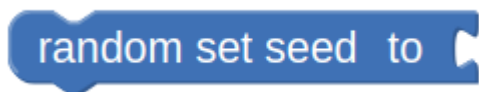
Returns the result of the first number raised to the power of the second.



Returns a random integer value between the given values, inclusive. The order of the arguments doesn't matter.



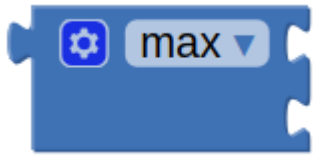
Returns a random value between 0 and 1.



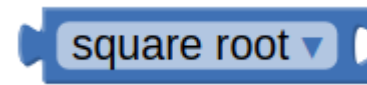
Use this block to generate repeatable sequences of random numbers. You can generate the same sequence of random numbers by first calling random set seed with the same value. This is useful for testing programs that involve random values.



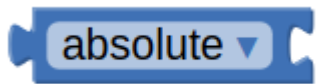
Returns the smallest value of a set of numbers. If there are unplugged sockets in the block, min will also consider 0 in its set of numbers. This block is a mutator and a dropdown.



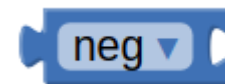
Returns the largest value of a set of numbers. If there are unplugged sockets in the block, max will also consider 0 in its set of numbers. This block is a mutator and a dropdown.



Returns the square root of the given number.



Returns the absolute value of the given number.



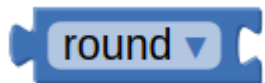
Returns the negative of a given number.



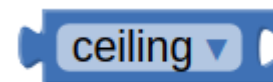
Returns the natural logarithm of a given number, that is, the logarithm to the base e (2.71828...).



Returns e (2.71828...) raised to the power of the given number.



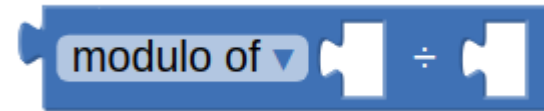
Returns the given number rounded to the closest integer. If the fractional part is $< .5$ it will be rounded down. If it is $> .5$ it will be rounded up. If it is exactly equal to $.5$, numbers with an even whole part will be rounded down, and numbers with an odd whole part will be rounded up. (This method is called round to even.)



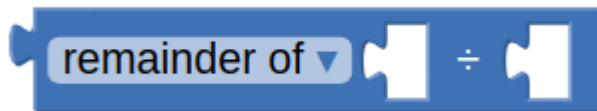
Returns the smallest integer that's greater than or equal to the given number.



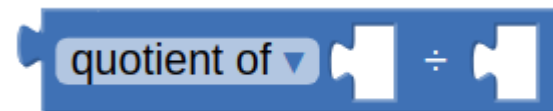
Returns the greatest integer that's less than or equal to the given number.



Modulo(a,b) is the same as remainder(a,b) when a and b are positive. More generally, modulo(a,b) is defined for any a and b so that $(\text{floor}(a/b) \times b) + \text{modulo}(a,b) = a$. For example, $\text{modulo}(11, 5) = 1$, $\text{modulo}(-11, 5) = 4$, $\text{modulo}(11, -5) = -4$, $\text{modulo}(-11, -5) = -1$. Modulo(a,b) always has the same sign as b, while remainder(a,b) always has the same sign as a.



Remainder(a,b) returns the result of dividing a by b and taking the remainder. The remainder is the fractional part of the result multiplied by b.



Returns the result of dividing the first number by the second and discarding any fractional part of the result.



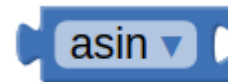
Returns the sine of the given number in degrees.



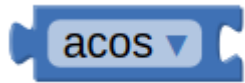
Returns the cosine of the given number in degrees.



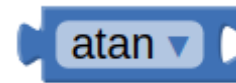
Returns the tangent of the given number in degrees.



Returns the arcsine of the given number in degrees.



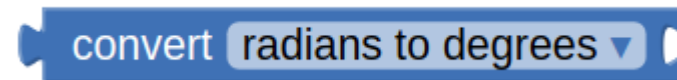
Returns the arccosine of the given number in degrees.



Returns the arctangent of the given number in degrees.



Returns the arctangent of y/x , given y and x .



Returns the value in degrees of the given number in radians. The result will be an angle in the range $[0, 360)$

convert degrees to radians ▾

Returns the value in radians of the given number in degrees. The result will be an angle in the range $[-\pi, +\pi)$

format as decimal number
places

Formats a number as a decimal with a given number of places after the decimal point. The number of places must be a non-negative integer. The result is produced by rounding the number (if there were too many places) or by adding zeros on the right (if there were too few).

is number? ▾

Returns true if the given object is a number, and false otherwise.

convert number binary to base 10 ▾

Takes a text string that represents a positive integer in binary and returns a string that represents the same number in decimal.

convert number base 10 to binary ▾

Takes a text string that represents a positive integer in decimal and returns a string that represents the same number in binary.

convert number base 10 to hex ▾

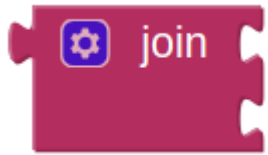
Takes a text string that represents a positive integer in decimal and returns a string that represents the same number in hexadecimal.

convert number hex to base 10 ▾

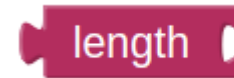
Takes a text string that represents a positive integer in hexadecimal and returns a string that represents the same number in decimal.



Contains a text string. This string can contain any characters (letters, numbers, or other special characters). On App Inventor it will be considered a Text object.



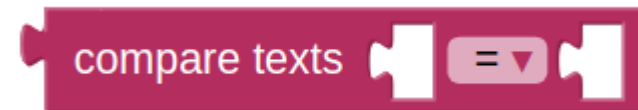
Appends all of the inputs to make a single string. If no inputs, returns an empty string.



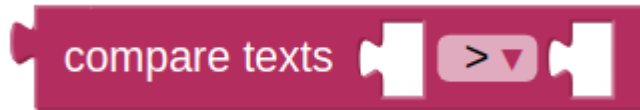
Returns the number of characters including spaces in the string. This is the length of the given text string.



Returns whether or not the string contains any characters (including spaces). When the string length is 0, returns true otherwise it returns false.



Returns whether or not the first string is lexicographically equal to the second string.



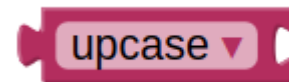
Returns whether or not the first string is lexicographically greater than the second string. A string is considered lexicographically greater than another if it is alphabetically greater than the other string. Essentially, it would come after it in the dictionary. All uppercase letters are considered smaller or to occur before lowercase letters. cat would be > Cat.



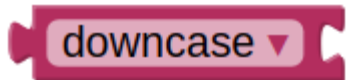
Returns whether or not the first string is lexicographically smaller than the second string. A string is considered lexicographically smaller than another if it is alphabetically smaller than the other string. Essentially, it would come after it in the dictionary. All uppercase letters are considered smaller or to occur before lowercase letters. Cat would be < cat.



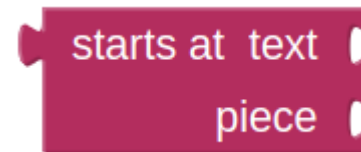
Removes any spaces leading or trailing the input string and returns the result.



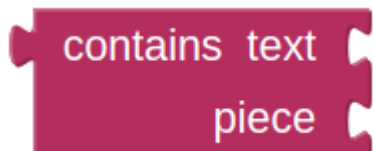
Returns a copy of its text string argument converted to all upper case



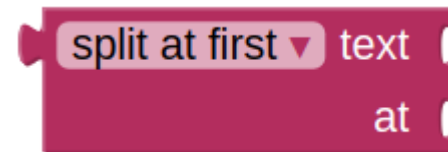
Returns a copy of its text string argument converted to all lower case



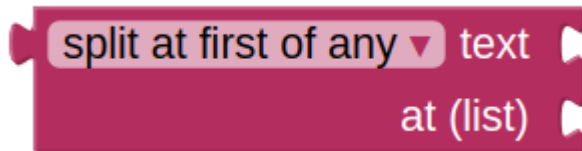
Returns the character position where the first character of piece first appears in text, or 0 if not present. For example, the location of ana in havana banana is 4.



Returns true if piece appears in text; otherwise, returns false.

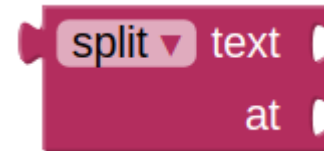


Divides the given text into two pieces using the location of the first occurrence of at as the dividing point, and returns a two-item list consisting of the piece before the dividing point and the piece after the dividing point. Splitting `apple,banana,cherry` with a comma as the splitting point returns a list of two items: the first is the text `apple` and the second is the text `banana,cherry`. Notice that the comma after `apple` doesn't appear in the result, because that is the dividing point.

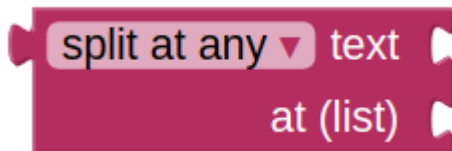


Divides the given text into a two-item list, using the first location of any item in the list at as the dividing point.

Splitting i love apples bananas apples grapes by the list [ba,ap] would result in a list of two items the first being i love and the second ples bananas apples grapes.

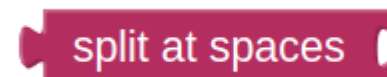


Divides text into pieces using at as the dividing points and produces a list of the results. Splitting one,two,three,four at , (comma) returns the list one two three four. Splitting one-potato,two-potato,three-potato,four at -potato, returns the list one two three four.

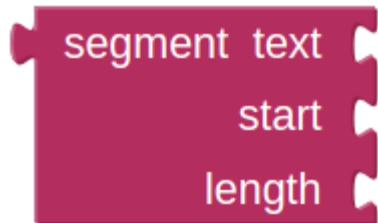


Divides the given text into a list, using any of the items in at as the dividing point, and returns a list of the results.

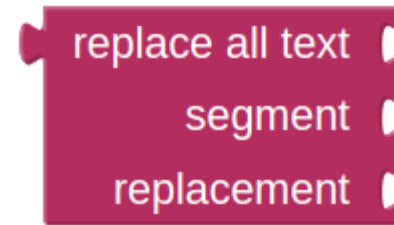
Splitting appleberry,banana,cherry,dogfood with at as the two-element list whose first item is a comma and whose second item is rry returns a list of four items: [applebe, banana, che, dogfood,]



Divides the given text at any occurrence of a space, producing a list of the pieces.



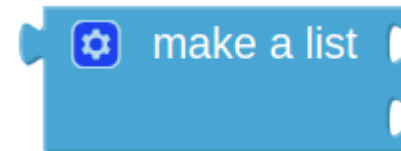
Extracts part of the text starting at start position and continuing for length characters.



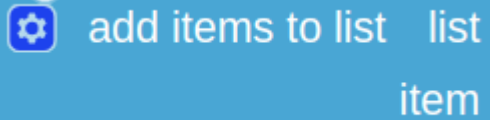
Returns a new text string obtained by replacing all occurrences of the substring with the replacement. Replace all with She loves eating. She loves writing. She loves coding as the text, She as the segment, and Hannah as the replacement would result in Hannah loves eating. Hannah loves writing. Hannah loves coding.



Creates an empty list with no elements.

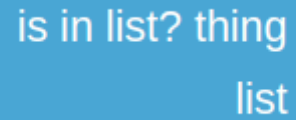


Creates a list from the given blocks. If you don't supply any arguments, this creates an empty list, which you can add elements to later. This block is a mutator. Clicking the blue plus sign will allow you to add additional items to your list.



A blue Scratch block with a gear icon on the left. The text "add items to list" is on the top line, "list" is on the bottom line, and "item" is on the bottom line to the right of "list".

Adds the given items to the end of the list. The difference between this and append to list is that append to list takes the items to be appended as a single list while add items to list takes the items as individual arguments. This block is a mutator.



A blue Scratch block with a notch on the left. The text "is in list?" is on the top line, "thing" is on the bottom line, and "list" is on the bottom line to the right of "thing".

If thing is one of the elements of the list, returns true; otherwise, returns false. Note that if a list contains sublists, the members of the sublists are not themselves members of the list. For example, the members of the list (1 2 (3 4)) are 1, 2, and the list (3 4); 3 and 4 are not themselves members of the list.



A blue Scratch block with a notch on the left and a bump on the right. The text "length of list" is on the top line, and "list" is on the bottom line.

Returns the number of items in the list



A blue Scratch block with a notch on the left and a bump on the right. The text "is list empty?" is on the top line, and "list" is on the bottom line.

If list has no items, returns true; otherwise, returns false.

pick a random item list

Picks an item at random from the list.

index in list thing
list

Returns the position of the thing in the list. If not in the list, returns 0.

select list item list
index

Selects the item at the given index in the given list.
The first list item is at index 1.

insert list item list
index
item

Inserts an item into the list at the given position


```
replace list item list  
index  
replacement
```

Inserts replacement into the given list at position index. The previous item at that position is removed.

```
remove list item list  
index
```

Removes the item at the given position.

```
append to list list1  
list2
```

Adds the items in the second list to the end of the first list.

```
copy list list
```

Makes a copy of a list, including copying all sublists.

is a list? thing

If thing is a list, returns true; otherwise, returns false.

list to csv row list

Interprets the list as a row of a table and returns a CSV (comma-separated value) text representing the Row. Each item in the row list is considered to be a field, and is quoted with double-quotes in the resulting CSV text. Items are separated by commas. For example, converting the list (a b c d) to a CSV row produces ("a", "b", "c", "d"). The returned row text does not have a line separator at the end.

list from csv row text

Parses a text as a CSV (comma-separated value) formatted row to produce a list of fields.
For example, converting ("a", "b", "c", "d") to a list produces (a b c d).

list to csv table list

Interprets the list as a table in row-major format and returns a CSV (comma-separated value) text representing the table. Each item in the list should itself be a list representing a row of the CSV table. Each item in the row list is considered to be a field, and is quoted with double-quotes in the resulting CSV text. In the returned text, items in rows are separated by commas and rows are separated by CRLF (\r\n).

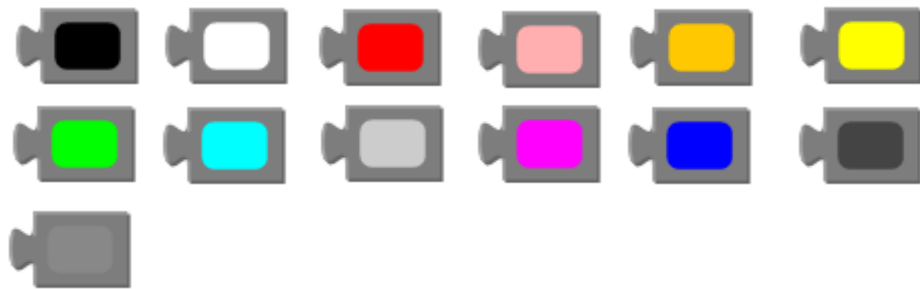
A blue Scratch block with a notch on the left and a bump on the right. The text inside reads "list from csv table text".

Parses a text as a CSV (comma-separated value) formatted table to produce a list of rows, each of which is a list of fields.

Rows can be separated by newlines (`\n`) or CRLF (`\r\n`).

A blue Scratch block with a notch on the left and a bump on the right. The text inside reads "look up in pairs key pairs notFound". To its right is a red Scratch block with a bump on the left and a notch on the right. The text inside reads "'not found'".


Used for looking up information in a dictionary-like structure represented as a list. This operation takes three inputs, a key, a list pairs, and a notFound result, which by default, is set to "not found". Here pairs must be a list of pairs, that is, a list where each element is itself a list of two elements. Lookup in pairs finds the first pair in the list whose first element is the key, and returns the second element. For example, if the list is ((a apple) (d dragon) (b boxcar) (cat 100)) then looking up 'b' will return 'boxcar'. If there is no such pair in the list, then the lookup in pairs will return the notFound result. If pairs is not a list of pairs, then the operation will signal an error.



This is a basic color block. It has a small square shape and has a color in the middle that represents the color stored internally in this block. If you click on the color in the middle, a pop-up appears on the screen with a table of 70 colors that you can choose from. Clicking on a new color will change the current color of your basic color block. Each basic color block that you drag from the Colors drawer to the Blocks Editor screen will display a table with the same colors when clicked.

A grey Scratch block with a notch on the left and a bump on the right. The text inside reads "make color". To its right is a blue Scratch block with a notch on the left and a bump on the right. The text inside reads "make a list". To the right of the "make a list" block are three input fields with values: 255, 0, and 0.

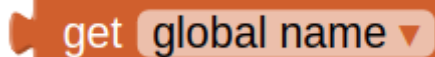
Make color takes in a list of 3 or 4 numbers. These numbers in this list represent values in an RGB code. RGB codes are used to make colors on the Internet. An RGB color chart is available here. This first number in this list represents the R value of the code. The second represents the G. The third represents the B. The fourth value is optional and represents the alpha value or how saturated the color is. The default alpha value is 100. Experiment with different values and see how the colors change using this block.

The image shows a Scratch 'split color' block. It is a dark grey block with a notch on the left and a bump on the right. The text 'split color' is written in white on the block.

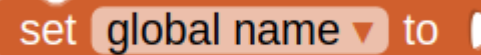
Split color does the opposite of make color. It takes in a color: a color block, variable holding a color, or property from one of the components representing a color and returns a list of the RGB values in that color's RGB code.

The image shows a Scratch 'initialize global name to' block. It is an orange block with a notch on the left and a bump on the right. The text 'initialize global name to' is written in white on the block.

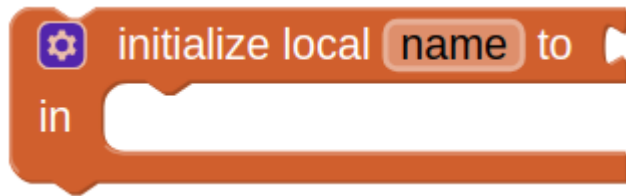
This block is used to create global variables. It takes in any type of value as an argument. Clicking on name will change the name of this global variable. Global variables are used in all procedures or events so this block will stand alone. Global variables can be changed while an app is running and can be referred to and changed from any part of the app even within procedures and event handlers. You can rename this block at any time and any associated blocks referring to the old name will be updated automatically.

The image shows a Scratch 'get global name' block. It is an orange block with a notch on the left and a bump on the right. The text 'get global name' is written in white on the block, with a small downward arrow next to 'global name'.

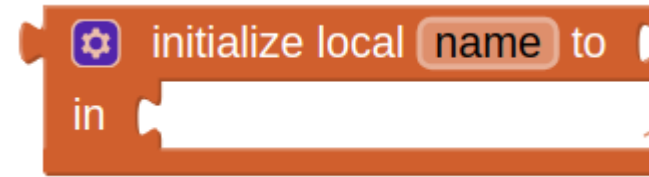
This block provides a way to get any variables you may have created.

The image shows a Scratch 'set global name to' block. It is an orange block with a notch on the left and a bump on the right. The text 'set global name to' is written in white on the block, with a small downward arrow next to 'global name'.

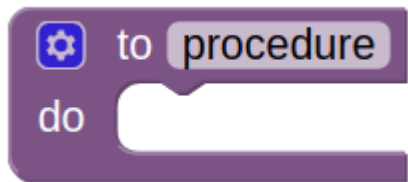
This block follows the same rules as get. Only variables in scope will be available in the dropdown. Once a variable *v* is selected, the user can attach a new block and give *v* a new value.



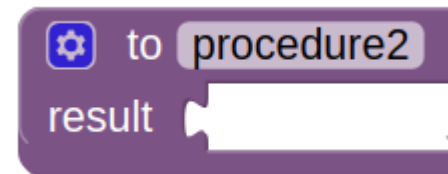
This block is a mutator that allows you to create new variables that are only used in the procedure you run in the DO part of the block. This way all variables in this procedure will all start with the same value each time the procedure is run. NOTE: This block differs from the block described below because it is a DO block. You can attach statements to it. Statements do things. That is why this block has space inside for statement blocks to be attached. You can rename the variables in this block at any time and any corresponding blocks elsewhere in your program that refer to the old name will be updated automatically



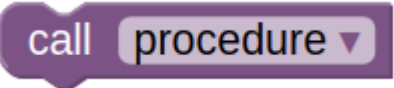
This block is a mutator that allows you to create new variables that are only used in the procedure you run in the RETURN part of the block. This way all variables in this procedure will all start with the same value each time the procedure is run. NOTE: This block differs from the block described above because it is a RETURN block. You can attach expressions to it. Expressions return a value. That is why this block has a socket for plugging in expressions. You can rename the variables in this block at any time and any corresponding blocks elsewhere in your program that refer to the old name will be updated automatically



Collects a sequence of blocks together into a group. You can then use the sequence of blocks repeatedly by calling the procedure. If the procedure has arguments, you specify the arguments by using the block's mutator button. If you click the blue plus sign, you can drag additional arguments into the procedure. When you create a new procedure block, App Inventor chooses a unique name automatically. You can click on the name and type to change it. Procedure names in an app must be unique. App Inventor will not let you define two procedures in the same app with the same name. You can rename a procedure at any time while you are building the app, by changing the label in the block. App Inventor will automatically rename the associated call blocks to match.

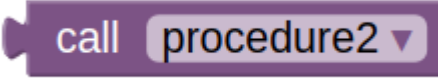


Collects a sequence of blocks together into a group. You can then use the sequence of blocks repeatedly by calling the procedure. If the procedure has arguments, you specify the arguments by using the block's mutator button. If you click the blue plus sign, you can drag additional arguments into the procedure. When you create a new procedure block, App Inventor chooses a unique name automatically. You can click on the name and type to change it. Procedure names in an app must be unique. App Inventor will not let you define two procedures in the same app with the same name. You can rename a procedure at any time while you are building the app, by changing the label in the block. App Inventor will automatically rename the associated call blocks to match. Calling this procedure returns a result.

A purple call block with a camera icon on the left, the text "call procedure", and a small downward arrow on the right.


call procedure ▾

When you create a procedure, App Inventor automatically generates a call block and places it in the Variables drawer. You use the call block to invoke the procedure.

A purple call block with a camera icon on the left, the text "call procedure2", and a small downward arrow on the right.

call procedure2 ▾

After creating this procedure, a call block that needs to be plugged in will be created. This is because the result from executing this procedure will be returned in that call block and the value will be passed on to whatever block is connected to the plug.

A red obfuscated text block with a camera icon on the left, the text "Obfuscated Text", a small square icon, and a double quote on the right.

Obfuscated Text " □ "

Produces text like a text block. The difference is that the text is not easily discover-able by examining the app's APK. Use when creating apps to distribute that include confidential information, for example, API keys. This provides only low level security against expert adversaries.



