App Inventor Glossary

Argument

Often in Computer Science, the inputs to procedures or events are called arguments. These arguments are local variables whose scope is inside that procedure or event.

Behavior

An app is said to have behavior. An app's behavior is how the app responds to user initiated and external events.

Block

App Inventor is a blocks programming language. Blocks are the pieces you connect together to tell your app what to do. They can be found in the Blocks Editor.

Blockly

Blockly is the name of the visual programming editor that App Inventor uses to make the blocks in the browser.

Blocks Editor

The screen found by clicking the **Blocks** button on the design screen. This is where you tell your app what to do.

Comment

Comments allow you to write reminders or quick remarks on blocks of code. You can use them to explain what certain blocks do or what you want to do later on. As comments are not run, they are for the user and not for the computer. Using comments can allow you or others to better understand your code when you come back to it later on. You can add or remove a comment by right-clicking on a block.

Component

Components are the pieces of your app that do actions for you. On the design screen, components are dragged from the Components Palette and placed on the phone. Examples of components are Label, Sound, or Button.

Designer

The screen where you can drag and drop component pieces and design them using the User Interface.

Drawer

The second box in the hierarchy of blocks that goes Palette to Drawer to Block. An example of a drawer is Control.

Dropdown

Some blocks have a small dropdown arrow to the right of the name of the block. You can click on this arrow to change the name and function of the block. The get block is an example of a dropdown. For more help on this topic, check out the dropdowns page.

Emulator

The name of the fake phone that appears on your computer if you don't have an Android device to work with is an emulator.

Event Driven

We say that an app is *event driven* because it depends on events to know what to do. You don't tell your app to wait until a text message before doing something else. Instead, by using event handlers, you tell your app that *when* an event occurs, perform this task. This prevents your phone from spending tons of time waiting for events to happen while stopping everything else to wait. With event handlers, the phone can continue to do what is was assigned to do unless an event handler interrupts. We say that the flow of the program is determined by events.

Getter

A Getter is the block found in the Variables drawer that says get with a dropdown next to it. This block is used to return a local or global variable.

List

Lists are used to store information. If you wanted to keep track of all of the usernames of people who use your application, you would want to store that information in a list. When items are added to a list, they are placed in a certain position in the list. The position of an item in a list is often called its index. In App Inventor, the first item in a list has an index of 1, the second has an index of 2, and so on.

Mutator

Some blocks have a white plus sign on them in a blue box. These blocks are called mutators. If you click on the plus sign, a bubble pops up with the block on the left representing your function and all of its inputs and the block on the right with the name of one of the inputs. You can drag this input block into the function block and then your function block will now take an additional input. List and max are examples of mutators. For more help on this topic, check out the mutators page.

Palette

The broadest/outer most box that holds drawers.

Procedure

A procedure is a set of instructions. In App Inventor, a procedure is a set of blocks under a procedure block. For more help on this topic, check out the procedures page.

Properties

Every component has properties that can be changed or initialized on the Designer screen under the Properties which are located on the right hand side. They can also be changed or used in the

Blocks view by using getter or setter blocks for properties. These blocks will say something like get/set Button1.Height.

Setter

A setter is another block found in the Variables drawer that says set dropdown to. This block is used to assign new values to both local and global variables.

Variable

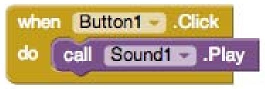
A variable is container that holds a value. There are two types of variables: global and local. For more information on this topic, check out the Global vs Local Variables page.

Important Concepts in App Inventor 2

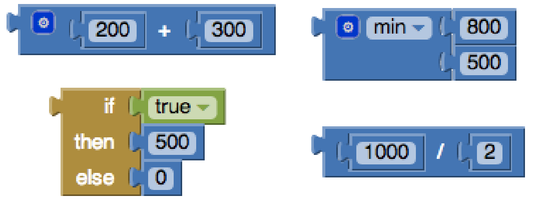
Commands and Expressions

When an event handler fires, it executes a sequence of commands in its body. A command is a block that specifies an action to be performed on the phone (e.g., playing sounds). Most command blocks are purple in color.

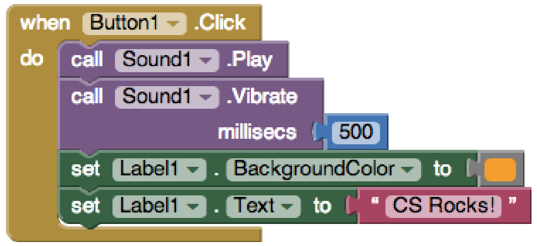
The Play block is an example of a command in HelloPurr:



Some commands require one or more **input** values (also known as parameters or arguments) to completely specify their action. For example, *call Sound1.Vibrate* needs to know the number of milliseconds to vibrate, *set Label1.BackgroundColor* needs to know the new background color of the label, and *set Label1.text* needs to know the new text string for the label. The need for input values is shown by sockets on the right edge of the command. These sockets can be filled with **expressions, blocks that denote a value**. Expression blocks have leftward-pointing plugs that you can imagine transmit the value to the socket. Larger expressions can be built out of simpler ones by horizontal composition. E.g., all of the following expressions denote the number 500:



Commands are shaped so that they naturally compose vertically into a command stack, which is just one big command built out of smaller ones. Here's a stack with four commands:



When this stack of commands are placed in a body of an event handler (e.g., the when.Button1.Click event handler), the command will be executed from the top to the bottom. If the stack of commands above is executed, then the phone will first play the sound, then vibrate, then change the label's color to be orange, and then label will show the text "CS rocks!" However, the execution works very fast: you would see all the actions happen at the same time.

Back to Main AI2 Concepts

Control Flow

When an event handler fires, you can imagine that it creates a karaoke-like control dot that flows through the command stack in its body. The control dot moves from the top of the stack to the bottom, and when it reaches a command, that command is executed -- i.e, the action of that command is performed. Thinking about control "flowing" through a program will help us understand its behavior.

The order of the commands, or the control flow is important when you make an app. You need to make sure which action should come first.

Back to Main AI2 Concepts

Arranging Components on the Screen

App components are organized vertically by default. In the Designer palette, using HorizontalArrangement and VerticalArrangement can allow you to change the organization of your components.

Back to Main AI2 Concepts

Manipulating Component State: Using Getters & Setters

Every component is characterized by various properties. What are some properties of a Label component? The current values of these properties are the state of the component. You can specify the initial state of a component in the Properties pane of the Designer window. App Inventor programs can get and set most component properties via blocks. E.g., the example in the "Commands" section above shows blocks for manipulating the state of Label1.

Getter blocks are expressions that get the current value of the property. Setter blocks are commands that change the value associated with the property. Some Label properties cannot be manipulated by blocks. Which ones?

Back to Main AI2 Concepts

Programming Your App to Make Decisions: Using Conditional Blocks

Sometimes you may want your app to perform different actions under different conditions. If you were making an app to hold all of the hours worked in the current week, you would need to test what day of the week it is to know where to store the hours.

To implement this in to your app, you would need to use conditionals. Conditionals refer to expressions or statements that evaluate to true or false.

Testing Conditionals with if and ifelse blocks

App Inventor provides two types of conditional blocks: if and ifelse, both of which are found in the Control drawer of the Built-In palette.

You can plug any Boolean expression into the “test” slot of these blocks. A Boolean expression is a mathematical equation that returns a result of either true or false. The expression tests the value of properties and variables using relational and logical operators such as the ones shown in the figure below:

For both if and ifelse, the blocks you put within the “then-do” slot will only be executed if the test is true. For an if block, if the test is false, the app moves on to the blocks below it. If the ifelse test is false, the blocks within the “else-do” slot are performed.

Example

Get picture of blocks and write example using ifelse, booleans,expression, etc.

For additional help on this topic, check out Chapter 18 from *App Inventor: Create your own Android Apps* by Dave Wolber, Hal Abelson, Liz Looney, and Ellen Spertus.

Back to Main AI2 Concepts

Events and Event Handlers

Apps are event-driven. They don't perform a set of instructions in a pre-determined order, instead they react to events. Clicking a button, dragging your finger, or touching down on the screen are all events.

With App Inventor, all activity occurs in response to an event. Your app shouldn’t contain blocks outside of an event’s “when-do” block. For instance, the blocks in the figure below don’t make sense floating alone.

As events occur, the app reacts by calling a sequence of functions. A function is anything you can do to or with a component such as setting the background color of a button to blue or changing the text of a label. We call an event and the set of functions that are performed in response to it: an event handler.

Events can be divided into 2 different types: user-initiated and automatic. Clicking a button, touching or dragging the screen, and tilting the phone are user-initiated events.Sprites colliding with each other or with canvas edges are automatic events.

For additional help on this topic, check out Chapter 14 from *App Inventor: Create your own Android Apps* by Dave Wolber, Hal Abelson, Liz Looney, and Ellen Spertus.

Back to Main AI2 Concepts

Using Multiple Screens in One App

In App Inventor, you can have one screen open a second screen. Later, the second screen can return to the screen that opened it. You can have as many screens as you like, but each screen

closes by returning to the screen that opened it. The screens can share information by passing and returning values when they open and close.
Building an app with multiple screens is a lot like creating several individual apps. Every screen that you create has its own components in the Designer window. In the Blocks Editor, you will be able to see only the components of the screen currently selected in the Designer. Similarly, the blocks of code related to a screen cannot refer to blocks of code in another screen. For more information, See the Colored Dots App Tutorial which explains multiple screens in detail.

Back to Main AI2 Concepts

PseudoRandom Number Generator and Random Set Seed

A pseudorandom number generator is an algorithm for generating a sequence of numbers that approximates the properties of random numbers. A random seed is a number or a vector that is chosen and used to initialize this number generator. By choosing different random seeds, your algorithm will choose random numbers in a slightly different way. Choosing a unique seed will return a unique random number sequence.

What this means is that if you continually use the same seed and use choose random item for a large amount of tests and data, you won't get as diverse or truly random results as if you chose a new seed each time.

Back to Main AI2 Concepts

Data & Databases

A database is a place where information or data can be stored until it is removed or replaced. Facebook uses a database to store usernames and corresponding passwords. Android devices have internal databases that store information about you or your phone. App Inventor allows us to access this database through the use of TinyDB.

App Inventor makes it easy to store data through its TinyDB and TinyWebDB components. Data is always stored as a tag-value pair, with the tag identifying the data for later retrieval. TinyDB should be used when it is appropriate to store data directly on the device. When data needs to be shared across phones (e.g., for a multiplayer game or a voting app), you'll need to use TinyWebDB instead. TinyWebDB is more complicated because you need to set up a callback procedure (the GotValue event handler), as well as a web database service.

To create your own web service, follow the instructions on the TinyWebDB component page.

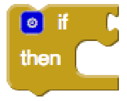
For additional help on this topic, check out Chapter 22 from *App Inventor: Create your own Android Apps* by Dave Wolber, Hal Abelson, Liz Looney, and Ellen Spertus.

Back to Main AI2 Concepts

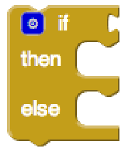
AI2 Control

Control Blocks

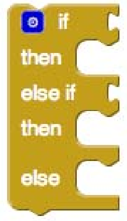
if & if else



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

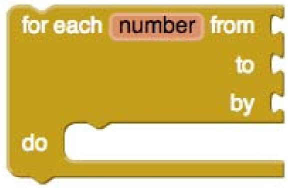


Tests a given condition. If the result is true, performs the actions in the -do sequence of blocks; otherwise, performs the actions in the -else sequence of blocks.



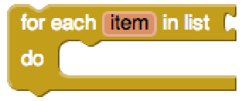
Tests a given condition. If the result is true, performs the actions in the -do sequence of blocks; otherwise tests the statement in the -else if section. If the result is true, performs the actions in the -do sequence of blocks; otherwise, performs the actions in the -else sequence of blocks.

for each from to



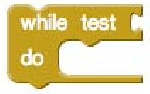
Runs the blocks in the do section for each numeric value in the range starting at *from* and ending at *to*, incrementing *number* by the value of *by* each time. Use the given variable name, *number* to refer to the current value. You can change the name *number* to something else if you wish.

for each in list



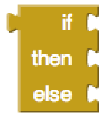
Runs the blocks in the do section for each item in the list. Use the given variable name, *item*, to refer to the current list item. You can change the name *item* to something else if you wish.

while



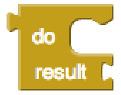
Tests the -test condition. If true, performs the action given in -do , then tests again. When test is false, the block ends and the action given in -do is no longer performed.

if then else



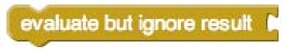
Tests a given condition. If the statement is true, performs the actions in the then-return sequence of blocks and returns the then-return value; otherwise, performs the actions in the else-return sequence of blocks and returns the else-return value.

do



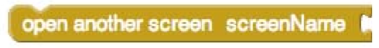
Sometimes in a procedure or another block of code, you may need to do something and return something, but for various reasons you may choose to use this block instead of creating a new procedure.

evaluate but ignore result



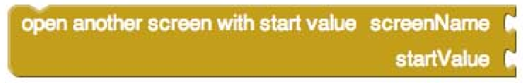
Provides a "dummy socket" for fitting a block that has a plug on its left into a place where there is no socket, such as one of the sequence of blocks in the do part of a procedure or an if block. The block you fit in will be run, but its returned result will be ignored. This can be useful if you define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

open another screen



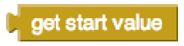
Opens the screen with the provided name.

open another screen with start value



Opens another screen and passes a value to it.

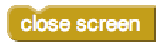
get start value



Returns the start value given to the current screen.

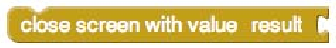
This value is given from using open another screen with start value or close screen with value.

close screen



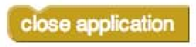
Closes the current screen.

close screen with value



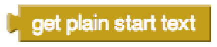
Closes the current screen and returns a value to the screen that opened this one

close application



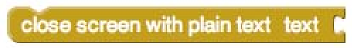
Closes the application.

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



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.

AI2 Logic

Logic Blocks

- true
- false
- not
- =
- ≠
- and
- or

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

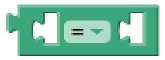
Performs logical negation, returning false if the input is true, and true if the input is false.

=

Tests whether its arguments are equal.

- Two numbers are equal if they are numerically equal, for example, 1 is equal to 1.0.
- Two text blocks are equal if they have the same characters in the same order, with the same case. For example, banana is not equal to Banana.
- Numbers and text are equal if the number is numerically equal to a number that would be printed with that text. For example, 12.0 is equal to the result of joining the first character of 1A to the last character of Teafor2.
- Two lists are equal if they have the same number of elements and the corresponding elements are equal.

Acts exactly the same as the = block found in Math



≠

Tests to see whether two arguments are not equal.

and

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 if true. You can consider this to be a logician's joke.

or

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.

AI2 Math

Math Blocks

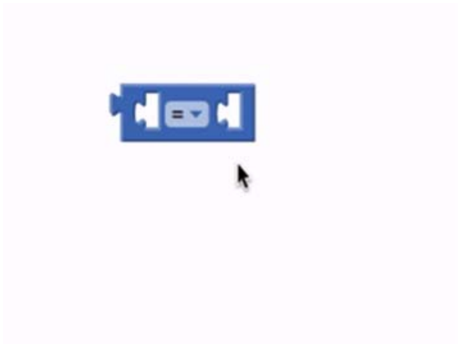
***Note: any Math blocks that have unplugged sockets will read the unplugged spot as a 0.*

- 0 (basic number block)
- =
- ≠,
- >,
- ≥,
- <,
- ≤,
- +
- -
- *
- /
- ^
- random integer
- random fraction
- random set seed to
- min
- max
- sqrt
- abs
- -
- log
- e^
- round
- ceiling
- floor
- modulo
- remainder
- quotient
- sin
- cos
- tan
- asin
- acos
- atan
- atan2
- convert radians to degrees
- convert degrees to radians
- format as a decimal
- is a number
- convert number

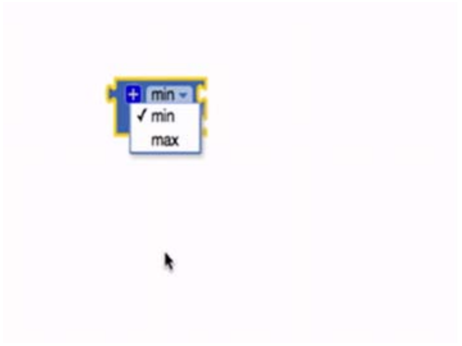
Can't find the math block you're looking for in the built-in blocks?

Some math blocks are dropdowns which means that they can be converted into different blocks. Here's a list of what is included in each dropdown:

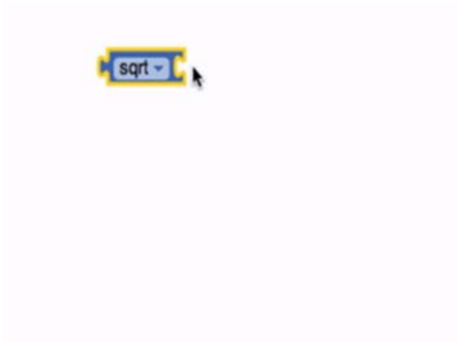
=, ≠, >, ≥, <, ≤



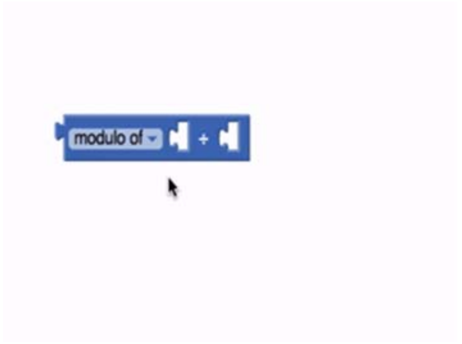
min, max



sqrt, abs, -, log, e^, round, ceiling, floor



modulo of, remainder of, quotient if



sin, cos, tan, asin, acos, atan

convert radians to degrees, convert degrees to radians

Basic Number Block

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.

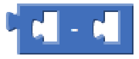
≤,

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

+

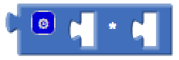
Returns the result of adding any amount of blocks that have a number value together. Blocks with a number value include the basic number block, length of list or text, variables with a number value, etc. This block is a mutator and can be expanded to allow more numbers in the sum.

-



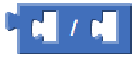
Returns the result of subtracting the second number from the first.

*



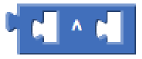
Returns the result of multiplying any amount of blocks that have a number value together. It is a mutator block and can be expanded to allow more numbers in the product.

/



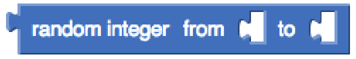
Returns the result of dividing the first number by the second.

^



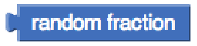
Returns the result of the first number raised to the power of the second.

random integer



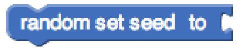
Returns a random integer value between the given values, inclusive. The order of the arguments doesn't matter.

random fraction



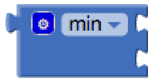
Returns a random value between 0 and 1.

random set seed to



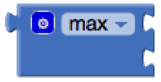
Use this block to generate repeatable sequences of random numbers. You can generate the same sequence of random numbers by first calling random set seed with the same value. This is useful for testing programs that involve random values.

min



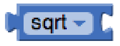
Returns the smallest value of a set of numbers. If there are unplugged sockets in the block, min will also consider 0 in its set of numbers. This block is a mutator and a dropdown.

max



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.

sqrt



Returns the square root of the given number.

abs

Returns the absolute value of the given number.

-

Returns the negative of a given number.

log

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

e^

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

round

Returns the given number rounded to the closest integer. If the fractional part is < .5 it will be rounded down. It 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*.)

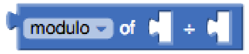
ceiling

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

floor

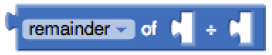
Returns the greatest integer that's less than or equal to the given number.

modulo



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 (floor(a/b)× b) + modulo(a,b) = a. For example, modulo(11, 5) = 1, modulo(-11, 5) = 4, modulo(11, -5) = -4, 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



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.

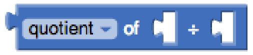
For example, remainder(11,5) = 1 because

11 / 5 = 2 ⅕

In this case, ⅕ is the fractional part. We multiply this by b, in this case 5 and we get 1, our remainder.

Other examples are remainder(-11, 5) = -1, remainder(11, -5) = 1, and remainder(-11, -5) = -1.

quotient



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

sin

Returns the sine of the given number in degrees.

cos

Returns the cosine of the given number in degrees.

tan

Returns the tangent of the given number in degrees.

asin

Returns the arcsine of the given number in degrees.

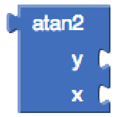
acos

Returns the arccosine of the given number in degrees.

atan

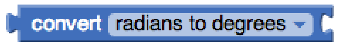
Returns the arctangent of the given number in degrees.

atan2



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

convert radians to degrees

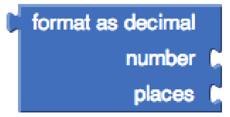


Returns the value in degrees of the given number in radians. The result will be an angle in the range [0, 360)

convert degrees to radians

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

format as decimal

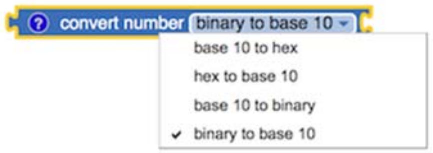


Formats a number as a decimal with a given number of places after the decimal point. The number of places must be a non-negative integer. The result is produced by rounding the number (if there were too many places) or by adding zeros on the right (if there were too few).

is a number

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

convert number

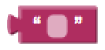


Takes a text string that represents a positive integer in one base and returns a string that represents the same number is another base. For example, if the input string is 10, then converting from base 10 to binary will produce the string 1010; while if the input string is the same 10, then converting from binary to base 10 will produce the string 2. If the input string is the same 10, then converting from base 10 to hex will produce the string A.

AI2 Text

Text Blocks

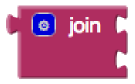
" "



Contains a text string.

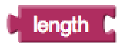
This string can contain any characters (letters, numbers, or other special characters). On App Inventor it will be considered a Text object.

join



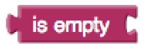
Appends all of the inputs to make a single string. If no inputs, returns an empty string.

length



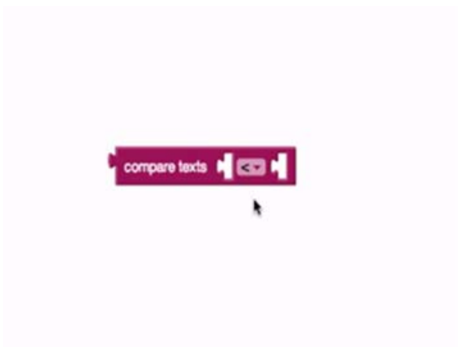
Returns the number of characters including spaces in the string. This is the length of the given text string.

is empty



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

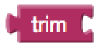
compare texts < > =



Returns whether or not the first string is lexicographically <, >, or = the second string depending on which dropdown is selected.

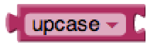
A string a considered lexicographically greater than another if it is alphabetically greater than the other string. Essentially, it would come after it in the dictionary. All uppercase letters are considered smaller or to occur before lowercase letters. *cat* would be > *Cat*.

trim



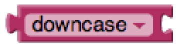
Removes any spaces leading or trailing the input string and returns the result.

upcase



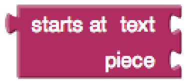
Returns a copy of its text string argument converted to all upper case

downcase



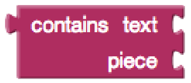
Returns a copy of its text string argument converted to all lower case

starts at



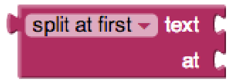
Returns the character position where the first character of *piece* first appears in text, or 0 if not present. For example, the location of *ana* in *havana banana* is 4.

contains



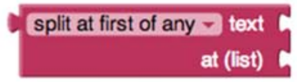
Returns true if *piece* appears in *text*; otherwise, returns false.

split at first



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,dogfood* 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,dogfood*. Notice that the comma after apple doesn't appear in the result, because that is the dividing point.

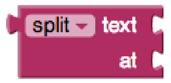
split at first of any



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

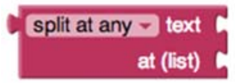
Splitting *i love apples bananas apples grapes* by the list [*ba*,*ap*] would result in a list of two items the first being *i love* and the second *ples bananas apples grapes*.

split



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*.

split at any



/sites/all/files/UserGuide/blocks/text/splitAtAny.png"

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.

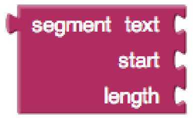
Splitting *appleberry,banana,cherry,dogfood* with at as the two-element list whose first item is a comma and whose second item is *rry* returns a list of four items: [*applebe, banana, che, dogfood,*]

split at spaces

split at spaces

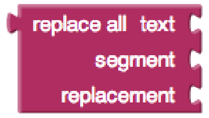
Divides the given text at any occurrence of a space, producing a list of the pieces.

segment



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

replace all



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*.

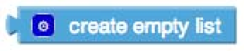
AI2 Lists

List Blocks

- create empty list
- make a list
- add items to list
- is in list
- length of list
- is list empty
- pick a random item
- index in list
- select list item
- insert list item
- replace list item
- remove list item
- append to list
- copy list
- is a list?
- list to csv row
- list to csv table
- list from csv row
- list from csv table
- lookup in pairs

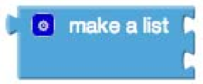
Need additional help understanding lists? Check out making lists on the Concepts page.

create empty list



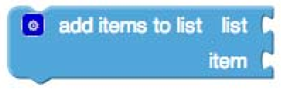
Creates an empty list with no elements.

make a list



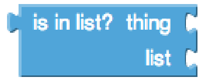
Creates a list from the given blocks. If you don't supply any arguments, this creates an empty list, which you can add elements to later.
This block is a mutator. Clicking the blue plus sign will allow you to add additional items to your list.

add items to list



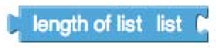
Adds the given items to the end of the list.
The difference between this and append to list is that append to list takes the items to be appended as a single list
while add items to list takes the items as individual arguments. This block is a mutator.

is in list?



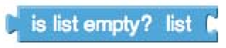
If thing is one of the elements of the list, returns true; otherwise, returns false.
Note that if a list contains sublists,
the members of the sublists are not themselves members of the list. For example, the members of the list (1 2 (3 4)) are 1, 2, and the list (3 4); 3 and 4 are not themselves members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

length of list



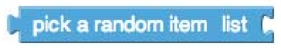
Returns the number of items in the list

is list empty?



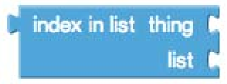
If list has no items, returns true; otherwise, returns false.

pick a random item



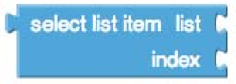
Picks an item at random from the list.

index in list



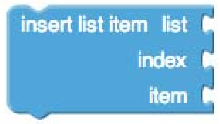
Returns the position of the *thing* in the list. If not in the list, returns 0.

select list item



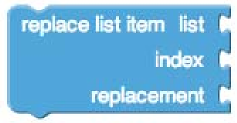
Selects the item at the given index in the given list. The first list item is at index 1.

insert list item



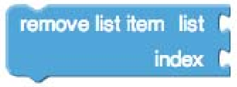
Inserts an item into the list at the given position

replace list item



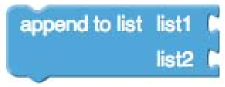
Inserts *replacement* into the given list at position index. The previous item at that position is removed.

remove list item



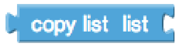
Removes the item at the given position.

append to list



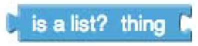
Adds the items in the second list to the end of the first list.

copy list



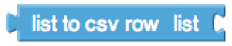
Makes a copy of a list, including copying all sublists.

is a list?



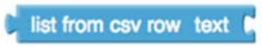
If *thing* is a list, returns true; otherwise, returns false.

list to csv row



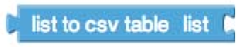
Interprets the list as a row of a table and returns a CSV (comma-separated value) text representing the row.
Each item in the row list is considered to be a field, and is quoted with double-quotes in the resulting CSV text. Items are separated by commas.
For example, converting the list (a b c d) to 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



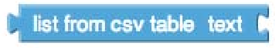
Parses a text as a CSV (comma-separated value) formatted row to produce a list of fields.
For example, converting ("a", "b", "c", "d") to a list produces (a b c d).

list to csv table



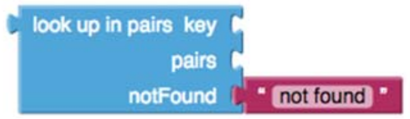
Interprets the list as a table in row-major format and returns a CSV (comma-separated value) text representing the table.
Each item in the list should itself be a list representing a row of the CSV table.
Each item in the row list is considered to be a field, and is quoted with double-quotes in the resulting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

list from csv table



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).

lookup in pairs



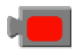
Used for looking up information in a dictionary-like structure represented as a list.
This operation takes three inputs, a *key*, a list *pairs*, and a *notFound* result, which by default, is set to "not found".
Here *pairs* must be a list of pairs, that is, a list where each element is itself a list 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.

AI2 Colors

There are three main types of color blocks:

- a color box
- make color
- split color

basic color blocks

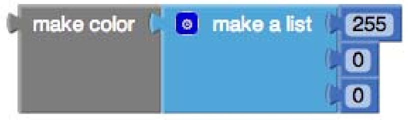


This is a basic color block. It has a small square shape and has a color in the middle that represents the color stored internally in this block.

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.

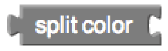
Each basic color block that you drag from the Colors drawer to the Blocks Editor screen will display a table with the same colors when clicked.

make color



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.

split color



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.

How do colors work in App Inventor?

Internally, App Inventor stores each color as a single number. When you use make color and take in a list as an argument, internally this list is then converted using App Inventor's color scheme and stored as a number. If you knew the numbers for the colors, you could even specify what color you wanted something to be by just setting its Color property to a specific number. If you want to see a chart of colors to numbers, check out this page.

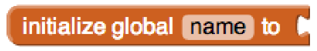
AI2 Variables

Variable Blocks

There are five main types of variable blocks:

- initialize global name to
- get
- set
- initialize local name to in (do)
- initialize local name to in (return)

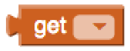
initialize global name to



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.

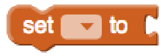
Global variables can be changed while an app is running and can be referred to and changed from any part of the app even within procedures and event handlers. You can rename this block at any time and any associated blocks referring to the old name will be updated automatically.

get



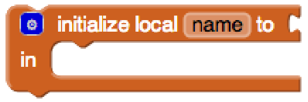
This block provides a way to get any variables you may have created.

set to



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.

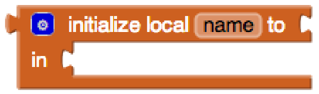
initialize Local name to - in (do)



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

initialize Local name to - in (return)



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.

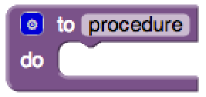
You can rename the variables in this block at any time and any corresponding blocks elsewhere in your program that refer to the old name will be updated automatically.

AI2 Procedures

A procedure is a sequence of blocks or code that is stored under a name, the name of your procedure block. Instead of having to keep putting together the same long sequence of blocks, you can create a procedure and just call the procedure block whenever you want your sequence of blocks to run. In computer science, a procedure also might be called a function or a method.

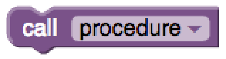
Procedure Blocks

procedure do



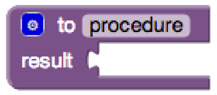
Collects a sequence of blocks together into a group. You can then use the sequence of blocks repeatedly by calling the procedure. If the procedure has arguments, you specify the arguments by using the block's mutator button. If you click the blue plus sign, you can drag additional arguments into the 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.

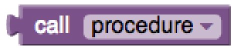


When you create a procedure, App Inventor automatically generates a call block and places it in the My Definitions drawer. You use the call block to invoke the procedure.

procedure result



Same as a procedure do block, but calling this procedure returns a result.



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.