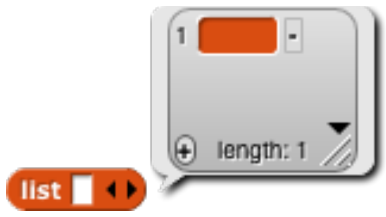
Exploring Tables with Snap*!*

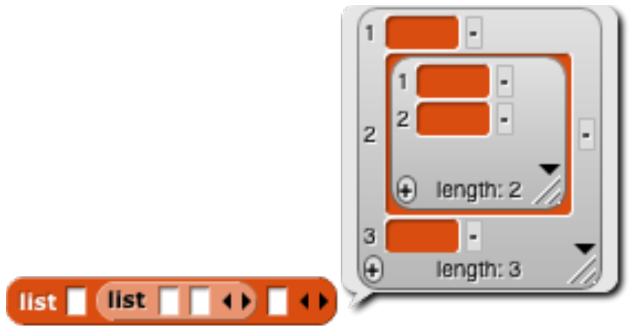
Jens Mönig
Feb. 24. 2016

Data often comes in the form of tables. Snap's answer to this is the generalization of lists. Because lists are first-class citizens of Snap, a table can be modeled as a list of lists, each sub-list representing a row, and same-indexed items of every row forming a logical column. Snap! version 4.0.5 introduces an alternative widget for exploring large lists and lists of lists.

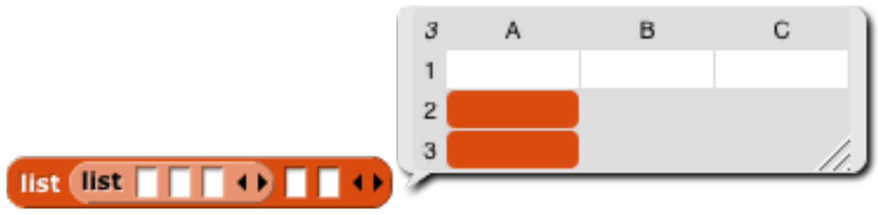
The usual widget for exploring a list is Snap's list watcher. It is modeled after Scratch's list watcher, providing a user-interface for exploring and directly editing a list. Since lists are first-class in Snap, list-watchers are not restricted to be shown onstage, but also appear inside sprites' speech bubbles and in result-balloons whenever the user clicks on a reporter in a scripting pane that returns a list:



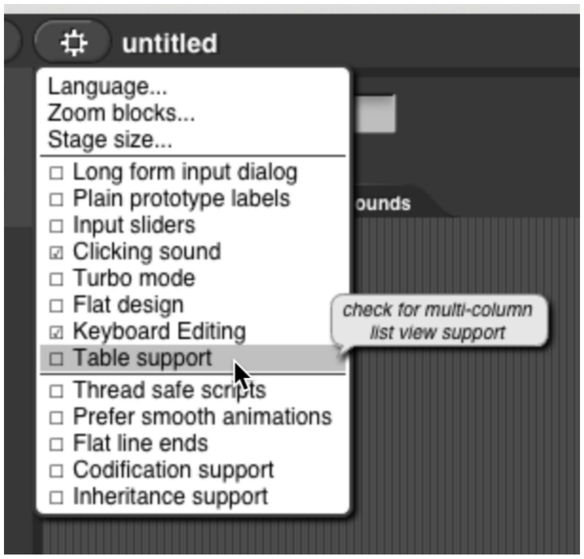
Likewise, lists within lists are usually shown in Snap as exactly that: A list watcher within another list watcher:



New in version 4.0.5 is that lists whose ***first*** item is another list are now displayed as tables:

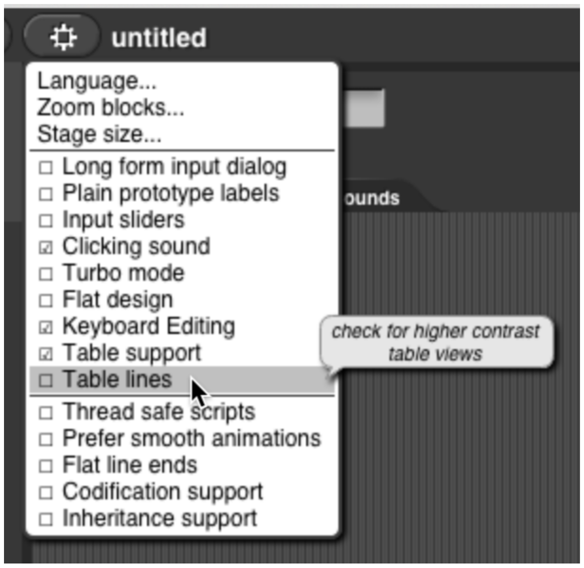


The new table view feature needs to be enabled in the settings menu (click on the gear button). Once enabled Snap remembers this preference across sessions. You can disable and re-enable support for tables again anytime.



A gridded layout of nested lists was first suggested by my friend and collaborator Brian Harvey back in the days of BYOB 3. Alas, I did not get around to implementing Brian's original idea until now.

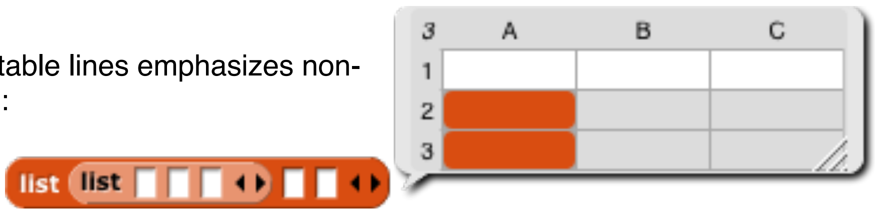
Table widgets are optimized to let users browse through large amounts of data. This is accomplished by simplifying the visual appearance of their components and by scrolling cell-wise as opposed to per-pixel sliding of list watchers. Unlike list watchers table widgets are "view-only" and do not enable direct editing of cells. Instead, tables can be manipulated using Snap's list blocks. Snap's Morphic architecture makes sure that any changes applied to the list elsewhere - either by directly editing a list or variable watcher, or through blocks and scripts - are immediately reflected in every table view for that list.



Note:

When Table support is enabled you get an additional choice in the preferences menu, that lets you add higher-contrast lines to table views. By default this setting is off in order to de-emphasize empty cells.

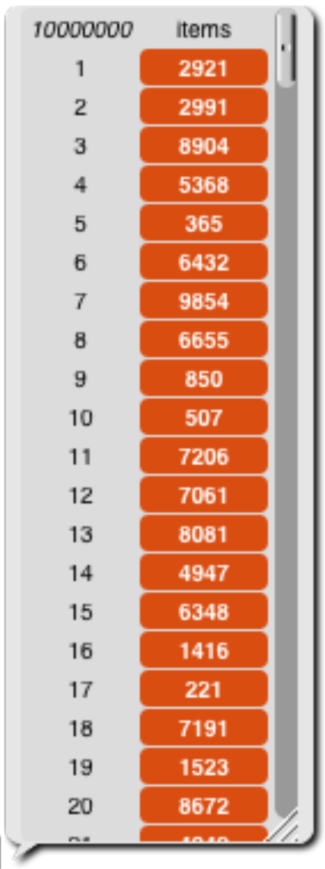
Conversely, enabling table lines emphasizes non-existing cells in tables:



CONTENTS

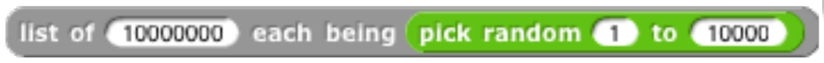
Large Lists

Since the new table widgets are more efficient at displaying large lists, Snap now automatically uses them whenever showing lists larger than 100 items, the current threshold for conventional list watchers, at which the user has to manually select another range of 100 items to show in the widget. The new table view is not constrained by this limit and lets the user seamlessly scroll through the whole list.



10000000	items
1	2921
2	2991
3	8904
4	5368
5	365
6	6432
7	9854
8	6655
9	850
10	507
11	7206
12	7061
13	8081
14	4947
15	6348
16	1416
17	221
18	7191
19	1523
20	8672

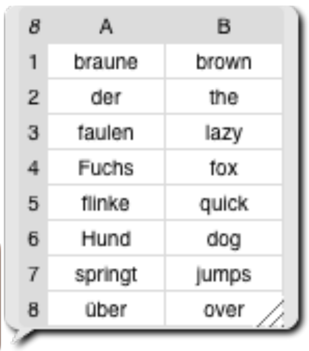
An example of a list containing 10 million random integers is shown to the right. Since the list is not 2-dimensional the widget's value-holding cells are colored in Snap's list category color and slightly rounded, like cells in list watchers. This emphasizes the single-dimensional list-ness of the structure.



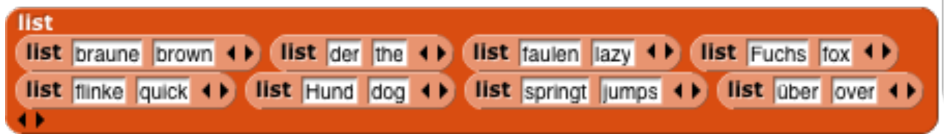
```
list of (10000000) each being (pick random (1) to (10000))
```

2D Lists

Two-dimensional lists are also automatically shown as tables. An example of a short and simple dictionary is shown here. The background color of the cells is white, same as the list-block's input slots. This coloring indicates that all cells can be safely accessed by their column and row indices.



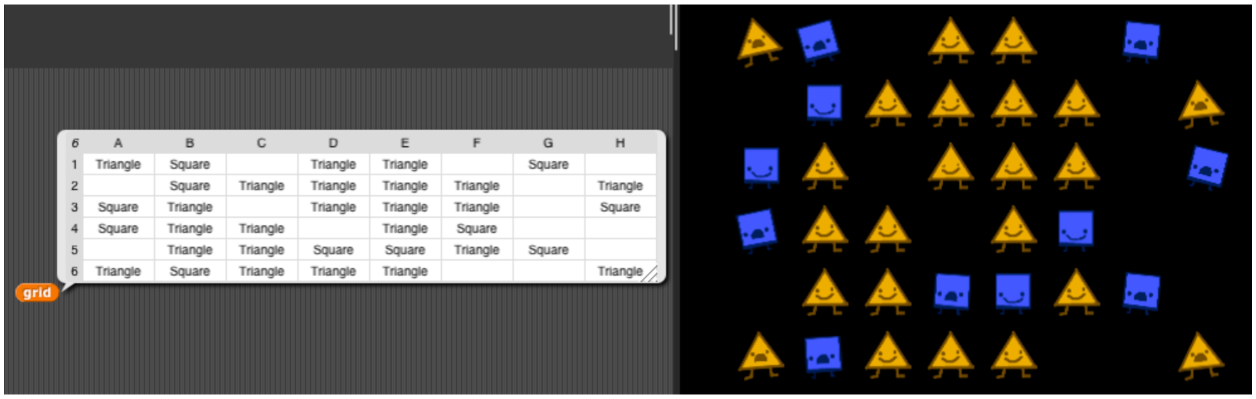
8	A	B
1	braune	brown
2	der	the
3	faulen	lazy
4	Fuchs	fox
5	flinke	quick
6	Hund	dog
7	springt	jumps
8	über	over



```
list
(list braune brown ◂▸) (list der the ◂▸) (list faulen lazy ◂▸) (list Fuchs fox ◂▸)
(list flinke quick ◂▸) (list Hund dog ◂▸) (list springt jumps ◂▸) (list über over ◂▸)
◂▸
```

Examples

Tables are sometimes convenient models for board-game type simulations. This Snap project mimics an aspect of Nicki Case's and Vi Hart's "Parable of the Polygons":



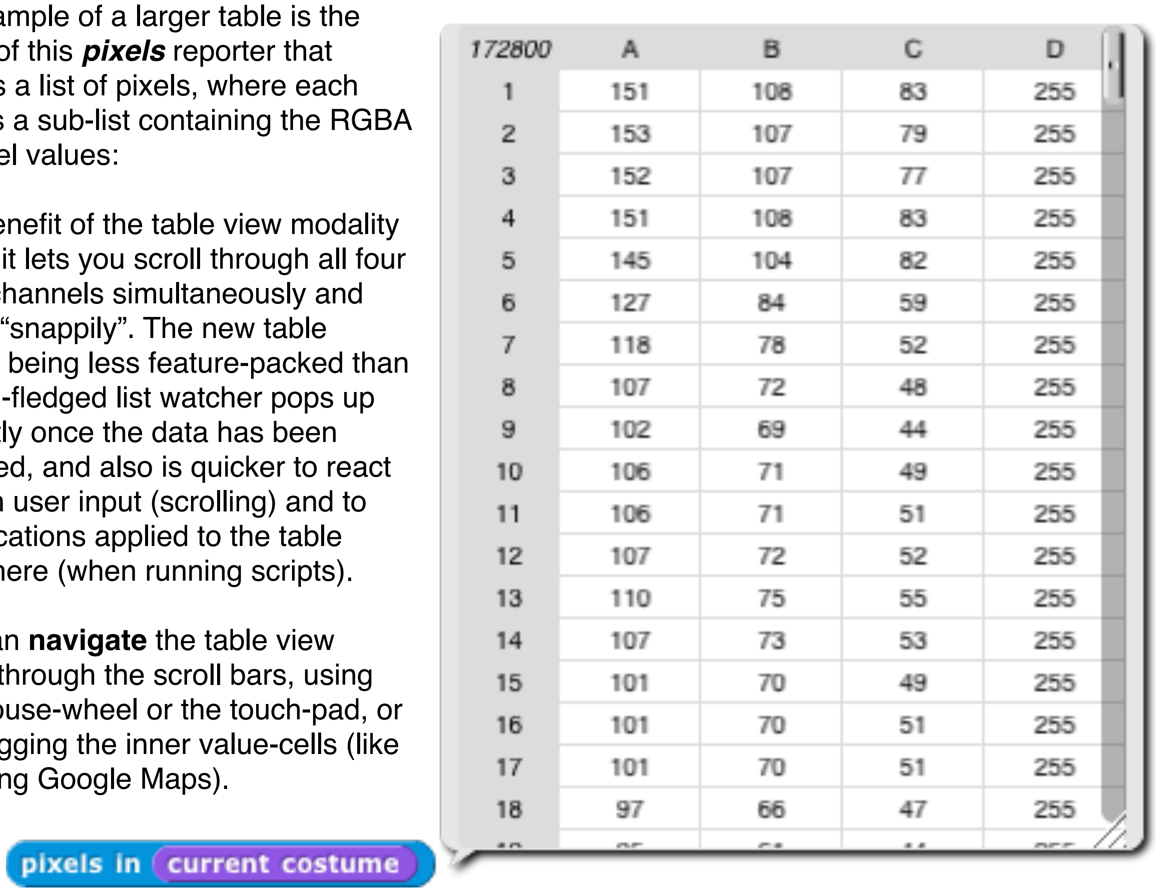
6	A	B	C	D	E	F	G	H
1	Triangle	Square		Triangle	Triangle		Square	
2		Square	Triangle	Triangle	Triangle	Triangle		Triangle
3	Square	Triangle		Triangle	Triangle	Triangle		Square
4	Square	Triangle	Triangle		Triangle	Square		
5		Triangle	Triangle	Square	Square	Triangle	Square	
6	Triangle	Square	Triangle	Triangle	Triangle			Triangle

The sprites - or rather clones - on the stage are basically a visualization of the underlying table data structure that is stored in a variable named ***grid*** here. It's fun to watch the table in the result-balloon inside the script editor change in synch with the pattern on the stage as the project is in running auto-solving mode.

An example of a larger table is the result of this ***pixels*** reporter that returns a list of pixels, where each pixel is a sub-list containing the RGBA channel values:

The benefit of the table view modality is that it lets you scroll through all four color channels simultaneously and rather "snappily". The new table widget being less feature-packed than the full-fledged list watcher pops up instantly once the data has been received, and also is quicker to react to both user input (scrolling) and to modifications applied to the table elsewhere (when running scripts).

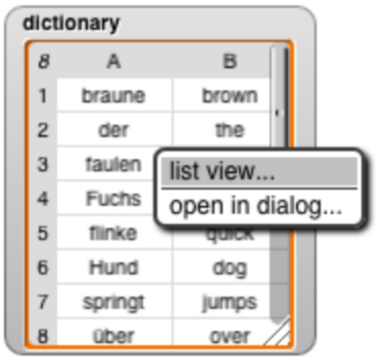
You can **navigate** the table view either through the scroll bars, using the mouse-wheel or the touch-pad, or by dragging the inner value-cells (like dragging Google Maps).



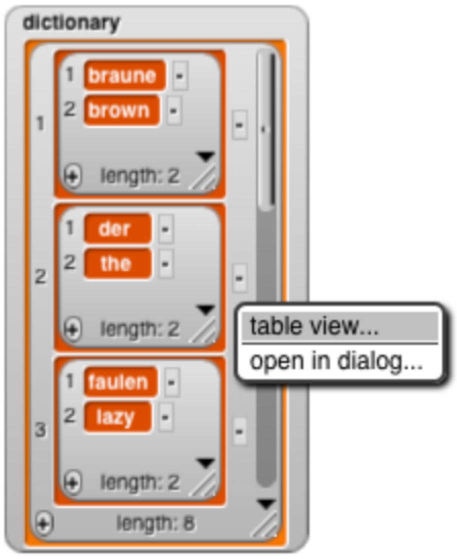
172800	A	B	C	D
1	151	108	83	255
2	153	107	79	255
3	152	107	77	255
4	151	108	83	255
5	145	104	82	255
6	127	84	59	255
7	118	78	52	255
8	107	72	48	255
9	102	69	44	255
10	106	71	49	255
11	106	71	51	255
12	107	72	52	255
13	110	75	55	255
14	107	73	53	255
15	101	70	49	255
16	101	70	51	255
17	101	70	51	255
18	97	66	47	255

Switching Views

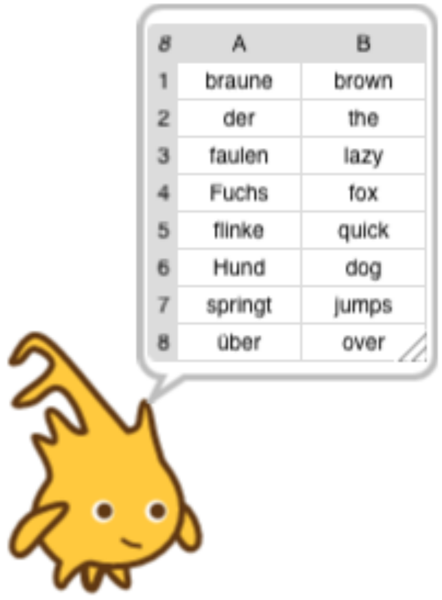
Table views are just another way to inspect and observe a list. You can switch from table view to list watcher and vice-versa using the context menu:



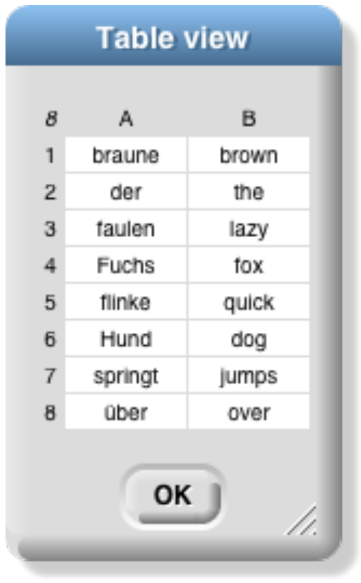
8	A	B
1	braune	brown
2	der	the
3	faulen	
4	Fuchs	
5	flinke	quick
6	Hund	dog
7	springt	jumps
8	über	over



You can now also inspect every list / table in a separate modeless dialog box outside of the stage, either using the context menu, or by double-clicking on a table view or list watcher:



8	A	B
1	braune	brown
2	der	the
3	faulen	lazy
4	Fuchs	fox
5	flinke	quick
6	Hund	dog
7	springt	jumps
8	über	over

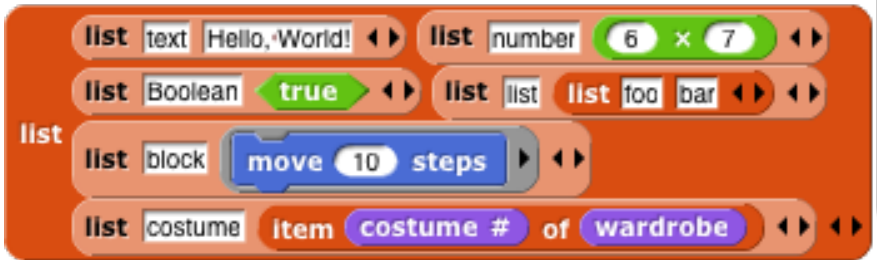


8	A	B
1	braune	brown
2	der	the
3	faulen	lazy
4	Fuchs	fox
5	flinke	quick
6	Hund	dog
7	springt	jumps
8	über	over

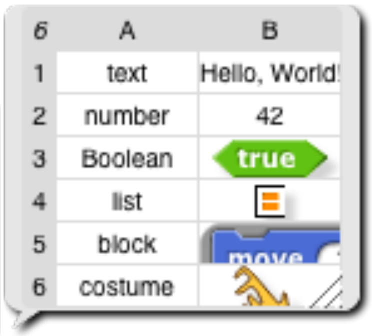
Within a table view dialog only table views are supported, i.e. of you double-click on a list watcher to open it in a dialog box it always appears as table view.

First-Class Data Types in Tables

Tables can hold any of Snap's first-class data objects. Currently these are text, numbers, Booleans, lists and rings (lambdafied blocks and scripts), and - experimentally - costumes:



```
list
  list text Hello, World!
  list number (6 × 7)
  list Boolean true
  list list (list foo bar)
  list block (move 10 steps)
  list costume (item costume # of wardrobe)
```

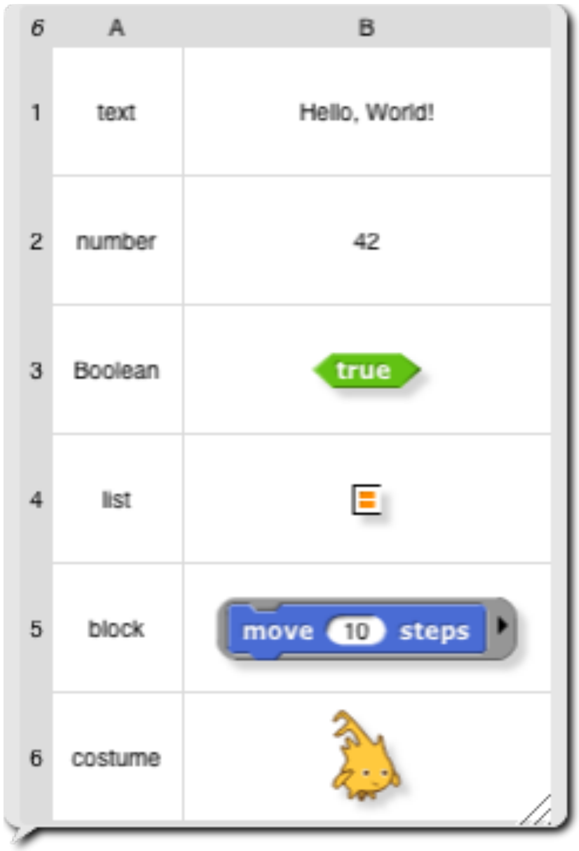


6	A	B
1	text	Hello, World!
2	number	42
3	Boolean	true
4	list	
5	block	move
6	costume	

Adjusting the Layout

Unlike list watchers the new table widgets don't automatically adjust cell-sizes to their values' visualizations. Instead they initially start out with a fixed default cell size for everything. This is one of the trade-offs for supporting views on large data sets.

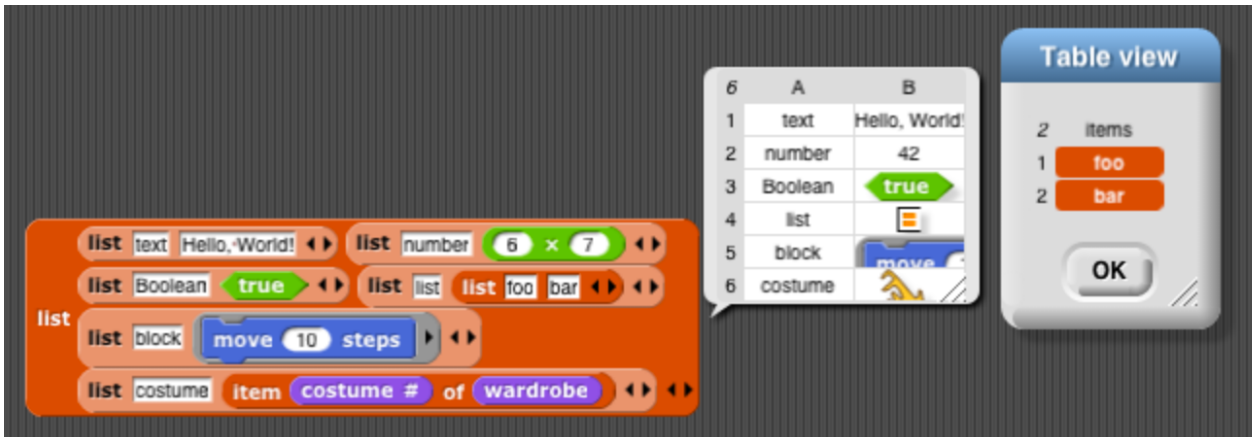
You can adjust the width of each column individually by **dragging the column-label left and right**. Holding the shift-key down while dragging any column-label globally changes the widths of all columns. Similarly you can increase or decrease row heights globally by **dragging any row label up or down**. this way users can explore diverse data:



6	A	B
1	text	Hello, World!
2	number	42
3	Boolean	true
4	list	
5	block	move 10 steps
6	costume	

Table Display Limitations

Another concession to enabling the user to scroll through large tables is only showing 2 data dimensions in a table view at one time. If in item in a table row contains another list, the cell does not offer an interactive, recursive list watcher but only shows the symbol for a list that is also used for list-type input slots in custom blocks. In this example the cell B4 hold a two-item list. It is shown symbolically in the table view:



6	A	B
1	text	Hello, World!
2	number	42
3	Boolean	true
4	list	☰
5	block	move
6	costume	

Table view

2	items
1	foo
2	bar

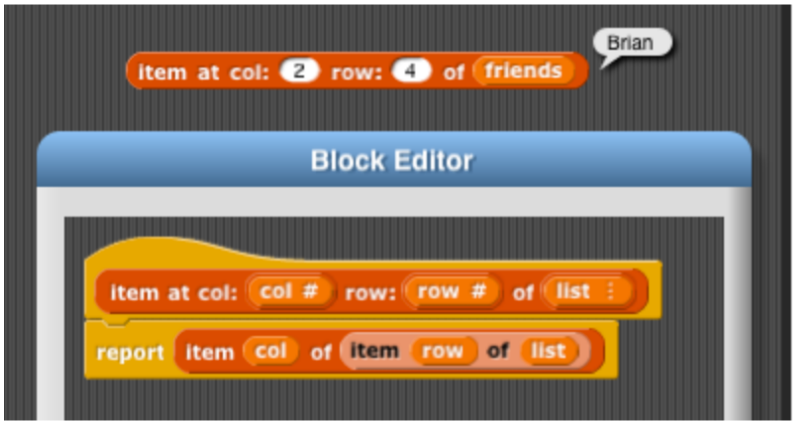
OK

Double clicking on a cell that holds a list opens a dialog box with a (table) view on the embedded list. This way you can explore more-dimensional lists and tables-within-tables

Display of text in table view cells is also limited to a single line of a few words, longer texts will be shown in abbreviated form. However, this only affects the *display* of text in cells, the actual *data* in the list is not altered in any way. Querying the item in the actual data structure using Snap’s list blocks always reports the full sized and correctly lined text object.

Blocks for Tables

The big idea behind tables in Snap is that there isn't any. Tables in Snap are nothing but lists of row-lists. Everything you already know about lists can be applied to tables. It's fun and very straightforward to build your own blocks for tables:

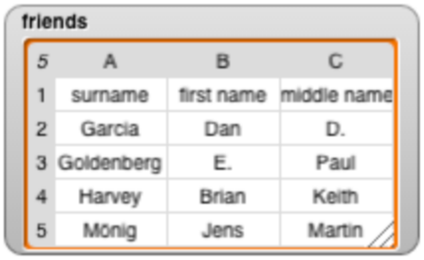


```
item at col: 2 row: 4 of friends
```

Brian

Block Editor

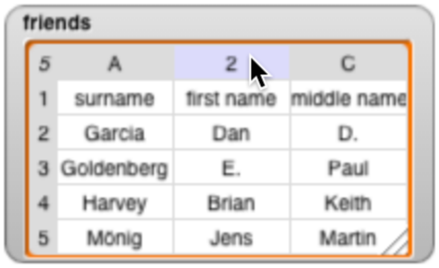
```
item at col: col # row: row # of list ⋮
report item col of item row of list
```



friends

5	A	B	C
1	surname	first name	middle name
2	Garcia	Dan	D.
3	Goldenberg	E.	Paul
4	Harvey	Brian	Keith
5	Mönig	Jens	Martin

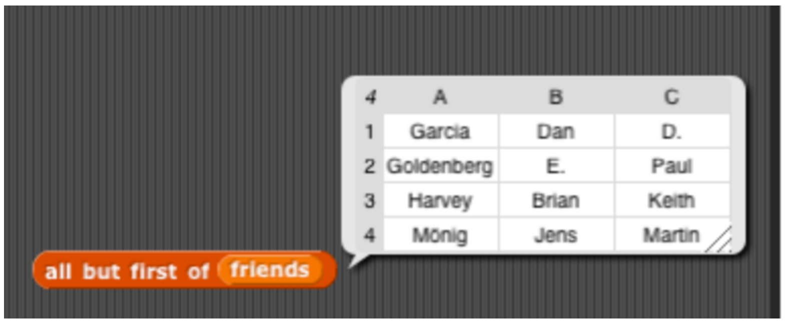
Note: Table views label rows by number and columns with letters. If you want to quickly find out the index number of a column instead, you can simply mouse over the column head, and it will be shown. Identifying the column number can be useful when accessing cells in tables with very many columns.



friends

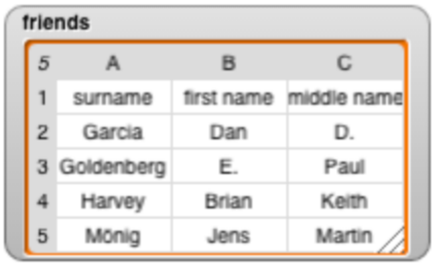
5	A	2	C
1	surname	first name	middle name
2	Garcia	Dan	D.
3	Goldenberg	E.	Paul
4	Harvey	Brian	Keith
5	Mönig	Jens	Martin

It's also fun and straightforward to directly use Snap's existing list blocks on tables, for example to strip the table of its first row, which often contains the column names:



```
all but first of friends
```

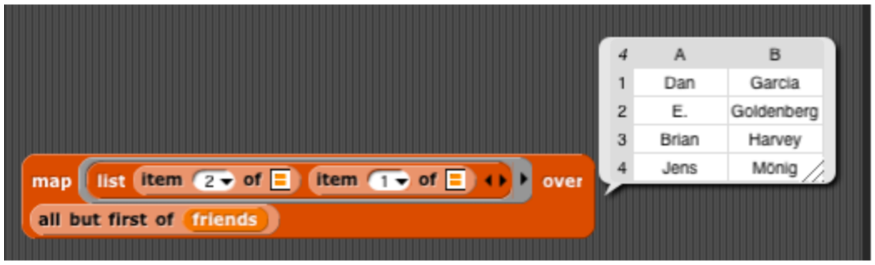
4	A	B	C
1	Garcia	Dan	D.
2	Goldenberg	E.	Paul
3	Harvey	Brian	Keith
4	Mönig	Jens	Martin



friends

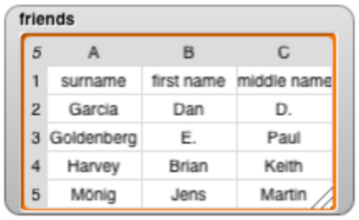
5	A	B	C
1	surname	first name	middle name
2	Garcia	Dan	D.
3	Goldenberg	E.	Paul
4	Harvey	Brian	Keith
5	Mönig	Jens	Martin

Likewise you can combine existing blocks for higher-order functions on tables. This example strips the table of its first row and last column, and also swaps the remaining two columns:



```
map (list (item (2) of ⊟) (item (1) of ⊟)) over
all but first of (friends)
```

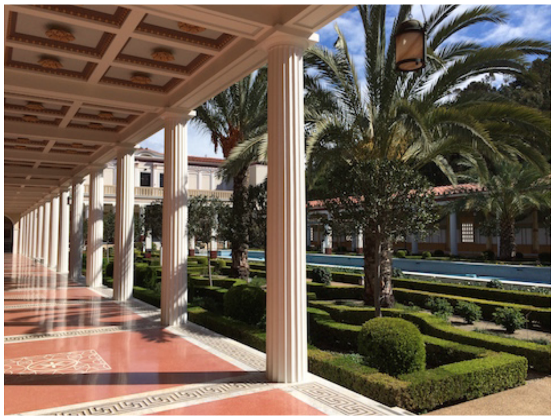
4	A	B
1	Dan	Garcia
2	E.	Goldenberg
3	Brian	Harvey
4	Jens	Mönig

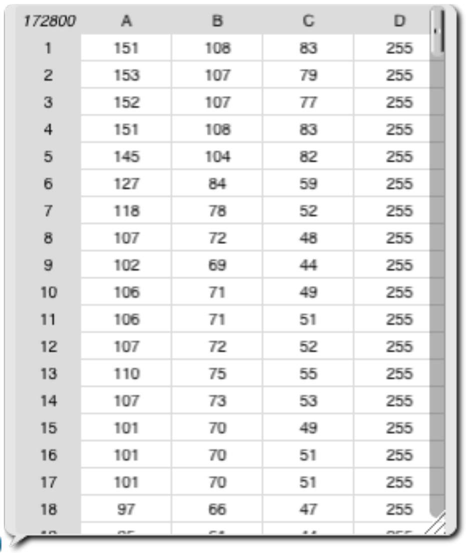


friends

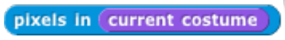
5	A	B	C
1	surname	first name	middle name
2	Garcia	Dan	D.
3	Goldenberg	E.	Paul
4	Harvey	Brian	Keith
5	Mönig	Jens	Martin

Rearranging a table by swapping columns already opens up all kinds of fun activities. Consider the pixel-data example from above, here shown alongside the image the pixel were extracted from:



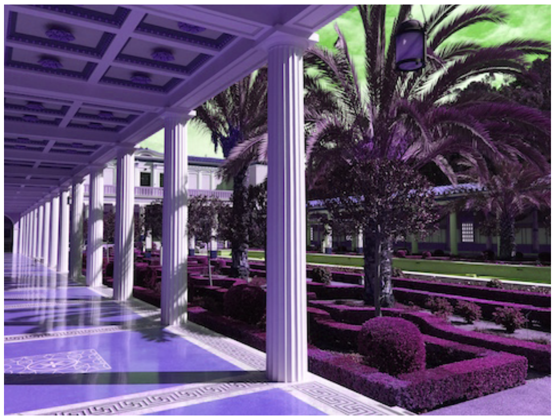


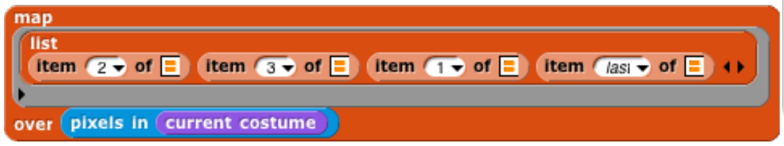
172800	A	B	C	D
1	151	108	83	255
2	153	107	79	255
3	152	107	77	255
4	151	108	83	255
5	145	104	82	255
6	127	84	59	255
7	118	78	52	255
8	107	72	48	255
9	102	69	44	255
10	106	71	49	255
11	106	71	51	255
12	107	72	52	255
13	110	75	55	255
14	107	73	53	255
15	101	70	49	255
16	101	70	51	255
17	101	70	51	255
18	97	66	47	255



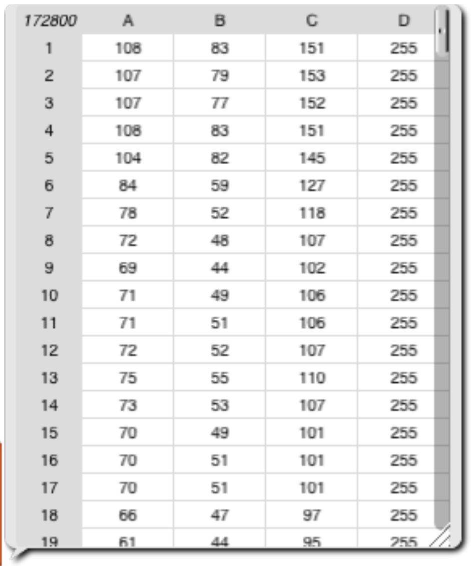
```
pixels in current costume
```

Swapping the color channels produces an interesting graphics effect:





```
map
  list item 2 of [] item 3 of [] item 1 of [] item last of []
over pixels in current costume
```



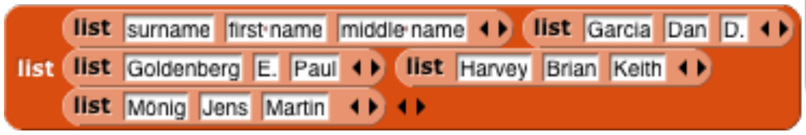
172800	A	B	C	D
1	108	83	151	255
2	107	79	153	255
3	107	77	152	255
4	108	83	151	255
5	104	82	145	255
6	84	59	127	255
7	78	52	118	255
8	72	48	107	255
9	69	44	102	255
10	71	49	106	255
11	71	51	106	255
12	72	52	107	255
13	75	55	110	255
14	73	53	107	255
15	70	49	101	255
16	70	51	101	255
17	70	51	101	255
18	66	47	97	255
19	61	44	95	255

Debugging Tables

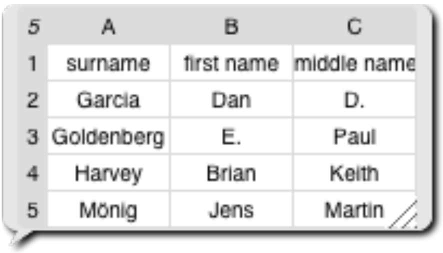
A downside of Snap's pedagogical idea to assemble tables out of lists of lists - rather than introducing another black-boxed first-class table object - is that it opens up ample opportunity for errors. If a table is not well-formed scripts operating on the assumption of certain table dimensions might trigger exceptions or silently produce wrong outcomes if an accessed cell does, in fact, not exist. Snap's new table view widget helps debug such errors by highlighting any quirks in the fabric of 2D lists that are assumed to resemble tables.

Well-Formed Tables

Snap assumes a list to be a table if the **first element of the list is another list longer than 1**. **The length of this first list** is assumed to be **the number of columns** in the table. The table is well-formed, if every other item in the list is also a list of exactly the same size as the first row. Well formed tables display a white background for every cell:



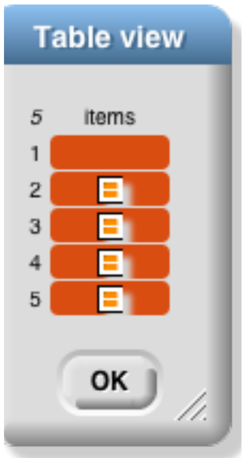
```
list (list surname first name middle name) (list Garcia Dan D.)
     (list Goldenberg E. Paul) (list Harvey Brian Keith)
     (list Mönig Jens Martin)
```



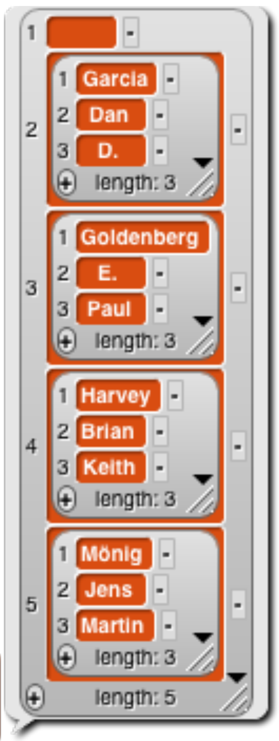
5	A	B	C
1	surname	first name	middle name
2	Garcia	Dan	D.
3	Goldenberg	E.	Paul
4	Harvey	Brian	Keith
5	Mönig	Jens	Martin

Missing First Row

A list whose first item isn't a list - or is a list of only length 1 - does ***not*** get recognized by Snap as being ***a table***. Therefore, by default, the conventional feature-rich list-watcher appears. If the list is over 100 items long, or if the user explicitly switches to "table view" in the list watcher's context menu, Snap displays the list inside the new table widget, but the table shows only a single column where each "row" is represented as a list symbol. In addition, all cells of the single column table are list-category colored and rounded to emphasize that Snap regards this table as a one-dimensional list.

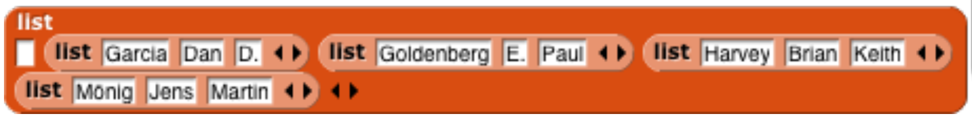


5	items
1	
2	☰
3	☰
4	☰
5	☰



```
1  [ ]
2  1 Garcia
   2 Dan
   3 D.
   length: 3
3  1 Goldenberg
   2 E.
   3 Paul
   length: 3
4  1 Harvey
   2 Brian
   3 Keith
   length: 3
5  1 Mönig
   2 Jens
   3 Martin
   length: 3
length: 5
```

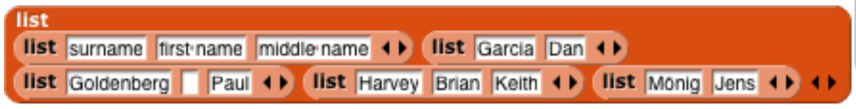
With the exception of the first row all custom blocks for tables and any list blocks combined for tables also work on such an "orphaned" table.



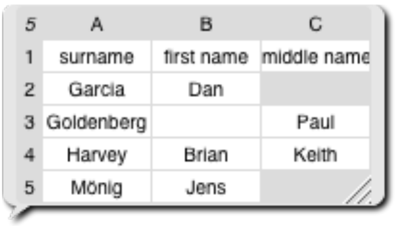
```
list [ ] (list Garcia Dan D.) (list Goldenberg E. Paul) (list Harvey Brian Keith)
     (list Mönig Jens Martin)
```

Incomplete Rows

In rows being ***shorter*** than the first one all unreachable cells are grayed out. In this example the rows 2 (Garcia) and 4 (Mönig) are both one item shorter than the first row. Since the cells C2 and C5 are unreachable they are both grayed out in the table view:



```
list
  list surname first name middle name
  list Garcia Dan
  list Goldenberg [] Paul
  list Harvey Brian Keith
  list Mönig Jens
```

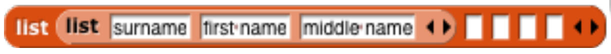


5	A	B	C
1	surname	first name	middle name
2	Garcia	Dan	▨
3	Goldenberg		Paul
4	Harvey	Brian	Keith
5	Mönig	Jens	▨

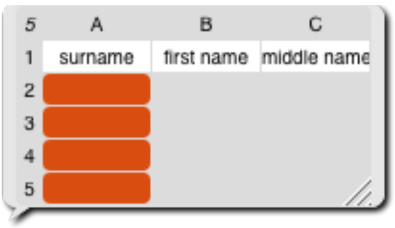
Note: "Unreachable" cells are not the same as "empty" cells. In the example above the cell B2 is empty, i.e. it does not hold any value. However that cell surely exists and can be reached. Therefore it is legitimately "white-listed".

Missing Rows

Snap regards a list whose first item is another list as table whose number of columns equals the length of the first row, and whose number of rows equals the length of the list itself. The following example is missing all rows but the first one. Therefore Snap considers it to be a table. The items of the first row are all reachable and thus "white-listed". Since all other items in the list are not lists themselves the cells B2 - B5 and C2 - C5 are unreachable und thus grayed out. The empty cells of the first column (A2 - A5) are list-category colored and rounded, indicating that this part of the "table" isn't actually a table at all but a single-dimensional list. Those cells can be reached, but only directly, not by specifying both a column and a row index.

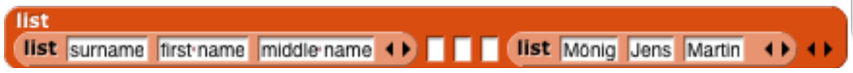


```
list (list surname first name middle name) [] [] [] []
```

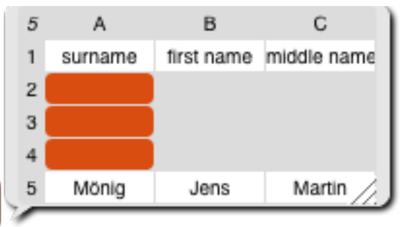


5	A	B	C
1	surname	first name	middle name
2	🟧	▨	▨
3	🟧	▨	▨
4	🟧	▨	▨
5	🟧	▨	▨

Likewise, the next example is missing three rows in the middle. Those three missing rows are indicated in the same way as the mostly empty table above, the cells of the first column indicating that this part of the table is single-dimensioned, and the unreachable cells of the other columns grayed out.



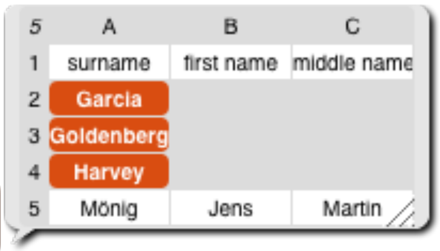
```
list
  list surname first name middle name
  [] [] []
  list Mönig Jens Martin
```



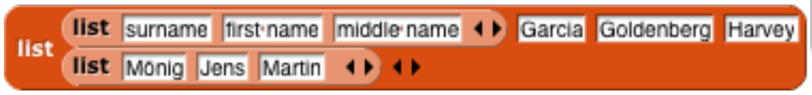
5	A	B	C
1	surname	first name	middle name
2	🟧	▨	▨
3	🟧	▨	▨
4	🟧	▨	▨
5	Mönig	Jens	Martin

Malformed Rows

In both of the “missing rows” examples above the items of the “outer” list can be accessed and replaced using “normal” list blocks. This way the missing rows could be added to the table. If, however, the “outer” list contains any item that isn’t a list, it still gets shown in the first column of the table view, but the cell is list-colored to indicate that a proper row element is missing.

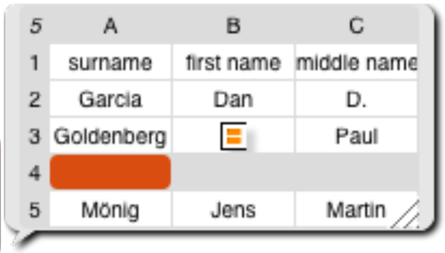


5	A	B	C
1	surname	first name	middle name
2	Garcia		
3	Goldenberg		
4	Harvey		
5	Mönig	Jens	Martin

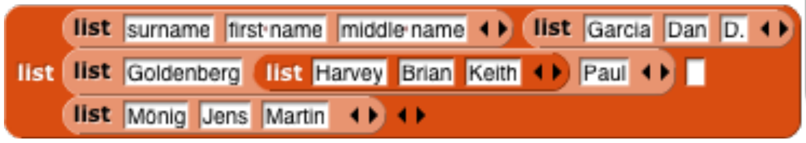


```
     list surname first name middle name ◂▸ Garcia Goldenberg Harvey
list
     list Mönig Jens Martin ◂▸ ◂▸
```

Another possible error source is wrong nesting of rows. In the following example the fourth row was accidentally dropped onto the second slot of the third row. Similar errors can occur when developing a parser for a new encoding. This error is visualized in the table view by an empty row and a list-symbol inside the middle cell of row 3. The user can double-click on the list-symbol cell to inspect that embedded list in a separate table view dialog box. That way its contents and possible sources of error can be discovered interactively.



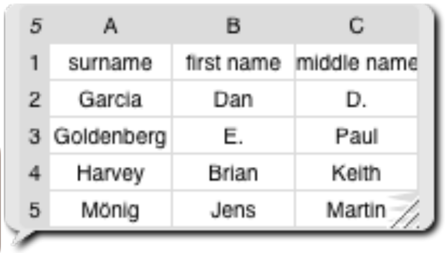
5	A	B	C
1	surname	first name	middle name
2	Garcia	Dan	D.
3	Goldenberg	≡	Paul
4			
5	Mönig	Jens	Martin



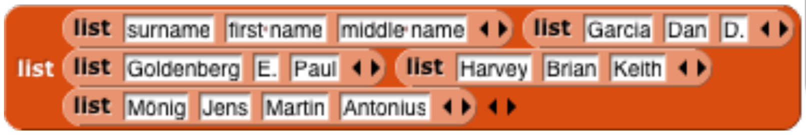
```
     list surname first name middle name ◂▸ list Garcia Dan D. ◂▸
list list Goldenberg list Harvey Brian Keith ◂▸ Paul ◂▸
     list Mönig Jens Martin ◂▸ ◂▸
```

Overshooting Rows

The pendant to an incomplete row is a row that is longer than the first row in the table. Consider the following example, where Jens Mönig has *two* middle names, but the structure of the table - defined by the length of its first row - only provides for one:

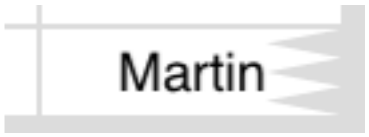


5	A	B	C
1	surname	first name	middle name
2	Garcia	Dan	D.
3	Goldenberg	E.	Paul
4	Harvey	Brian	Keith
5	Mönig	Jens	Martin



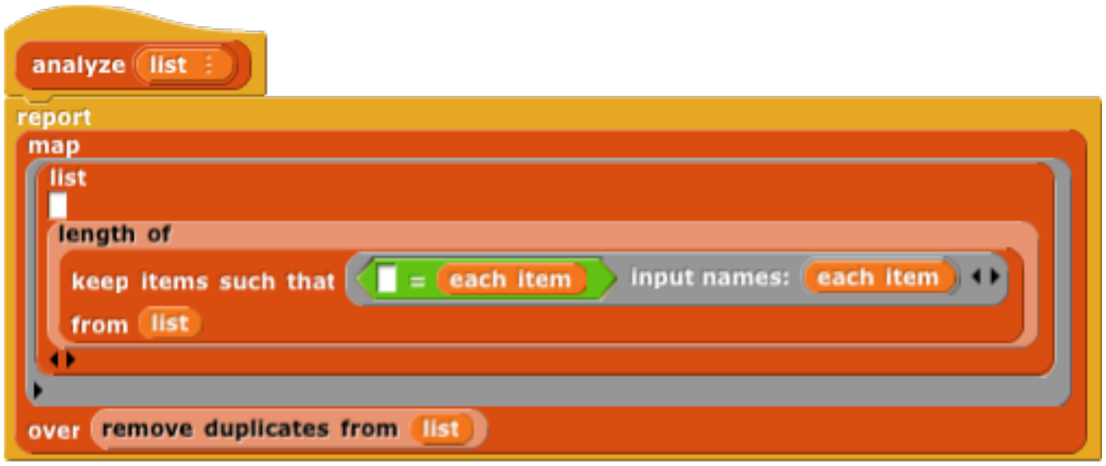
```
     list surname first name middle name ◂▸ list Garcia Dan D. ◂▸
list list Goldenberg E. Paul ◂▸ list Harvey Brian Keith ◂▸
     list Mönig Jens Martin Antonius ◂▸ ◂▸
```

Here, the rightmost cell of the overshooting row has a jagged right border to indicate that this row continues in a single dimension. The additional columns cannot be shown in the current table, they can only be inspected by switching to the conventional, more feature-rich list-watcher widget.



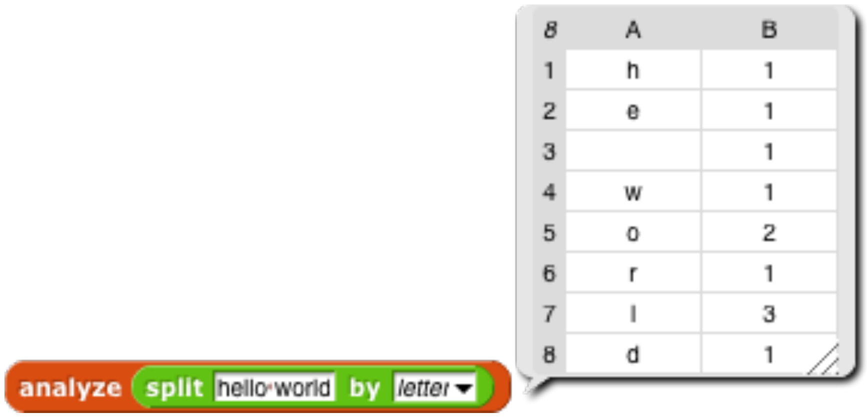
Analyzing and Transforming Data

When analyzing data a recurring theme is counting the occurrences of every unique item in a list. A fun and very useful block for this generic activity is the ANALYZE reporter. it reports a table that lists the frequency of each item in a given list. You can build it yourself using the list blocks from the tools and list libraries:



```
analyze list
report
  map
    list
      [ ]
      length of
        keep items such that ([ ] = each item) input names: each item
        from list
  over remove duplicates from list
```

When you analyze the text “hello world” (after splitting it up into a list of characters), you get a table with a row for every unique character. For each unique character the second column holds how often that character occurs in the source list:

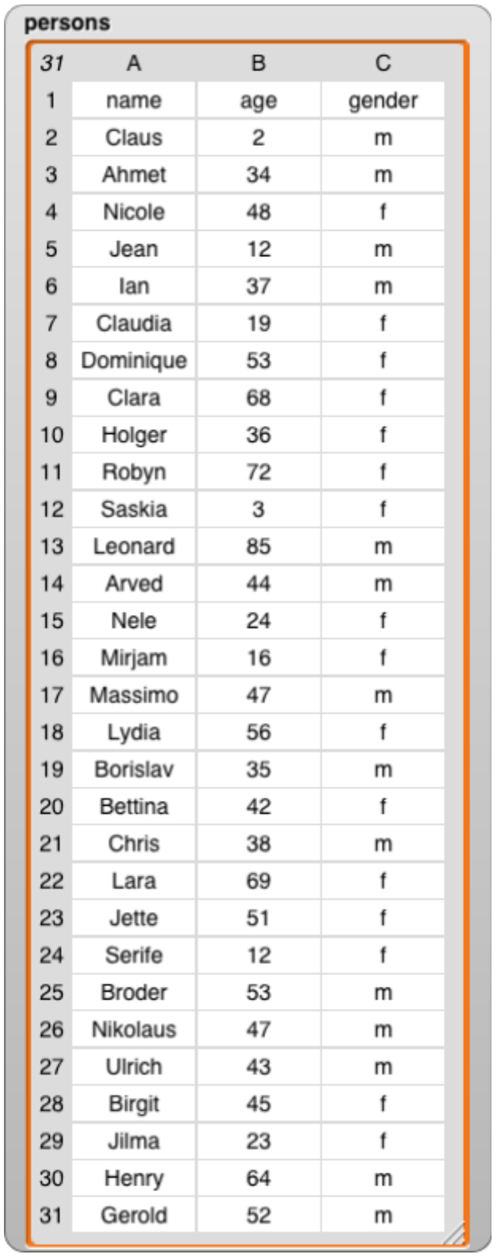


```
analyze split hello world by letter
```

8	A	B
1	h	1
2	e	1
3		1
4	w	1
5	o	2
6	r	1
7	l	3
8	d	1

As you can see, most character occur just once. However, the letter “o” is used twice in “hello world”, and the letter “l” occurs even three times.

Consider this table of persons:



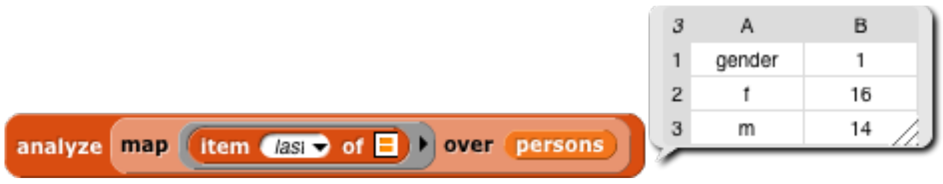
persons

31	A	B	C
1	name	age	gender
2	Claus	2	m
3	Ahmet	34	m
4	Nicole	48	f
5	Jean	12	m
6	Ian	37	m
7	Claudia	19	f
8	Dominique	53	f
9	Clara	68	f
10	Holger	36	f
11	Robyn	72	f
12	Saskia	3	f
13	Leonard	85	m
14	Arved	44	m
15	Nele	24	f
16	Mirjam	16	f
17	Massimo	47	m
18	Lydia	56	f
19	Borislav	35	m
20	Bettina	42	f
21	Chris	38	m
22	Lara	69	f
23	Jette	51	f
24	Serife	12	f
25	Broder	53	m
26	Nikolaus	47	m
27	Ulrich	43	m
28	Birgit	45	f
29	Jilma	23	f
30	Henry	64	m
31	Gerold	52	m

This is a short list of persons that stores each person’s name, age and gender. the table’s first row holds the column names. It is often a custom for the first row of a table to contain meta-information about the data, such as field names from the data base it was extracted from.

Example: Analyzing Gender Distribution

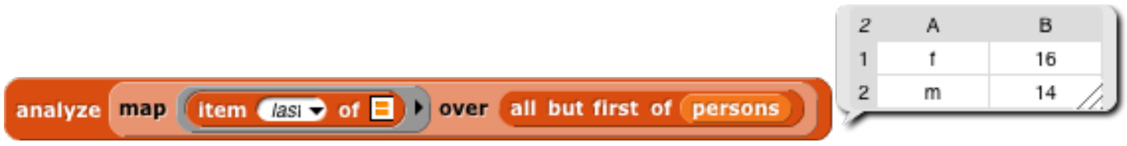
step 1:



```
analyze map (item (last) of ☐) over persons
```

3	A	B
1	gender	1
2	f	16
3	m	14

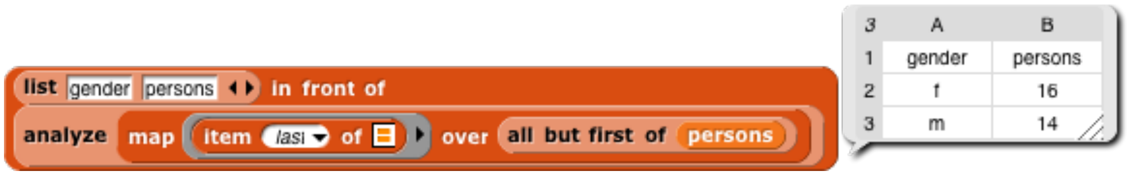
step 2: Ignoring the first row, which holds the column names



```
analyze map (item (last) of ☐) over all but first of persons
```

2	A	B
1	f	16
2	m	14

step 3: Adding new column names to the output

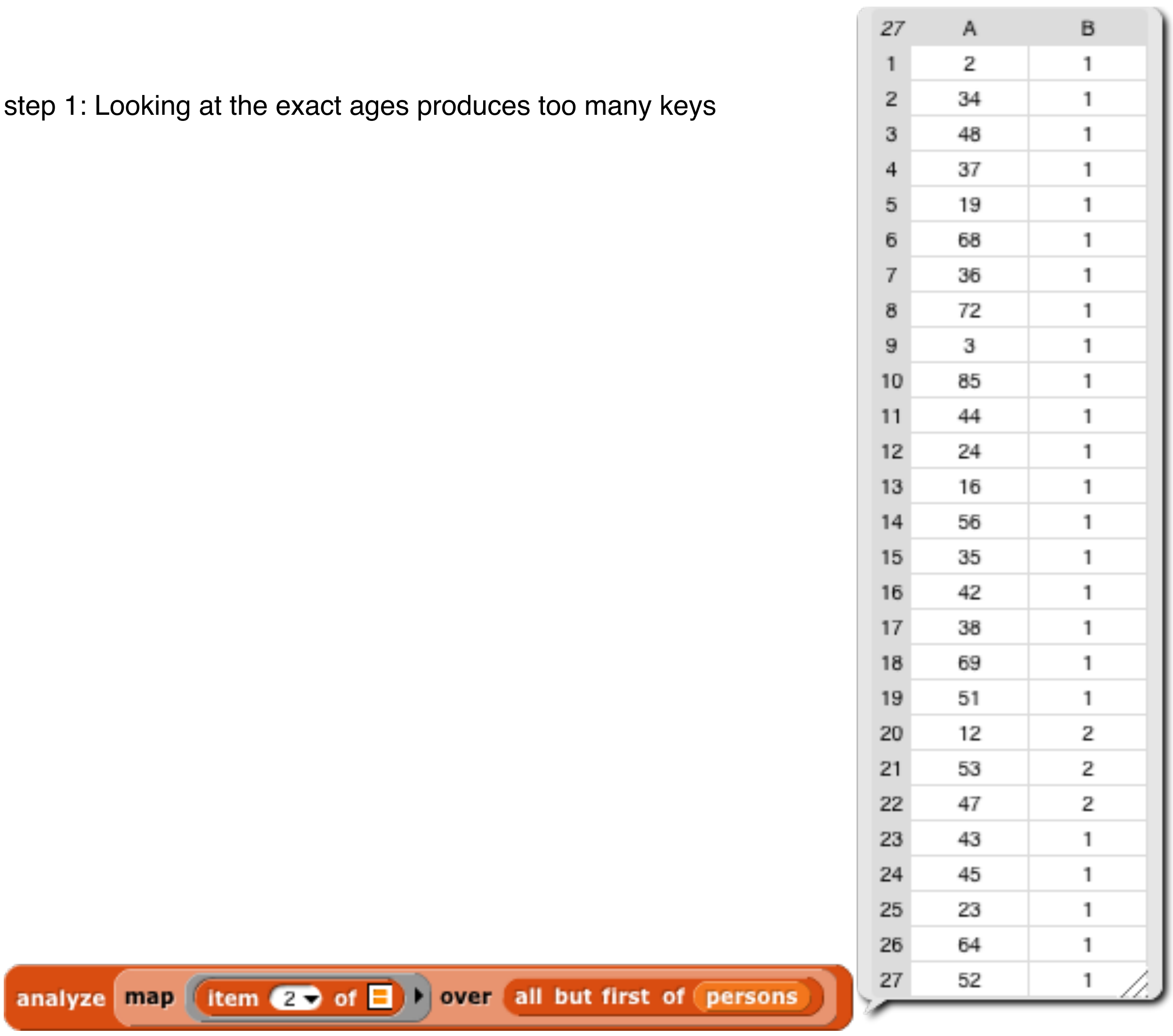


```
list gender persons in front of
analyze map (item (last) of ☐) over all but first of persons
```

3	A	B
1	gender	persons
2	f	16
3	m	14

Example: Analyzing Age Distribution

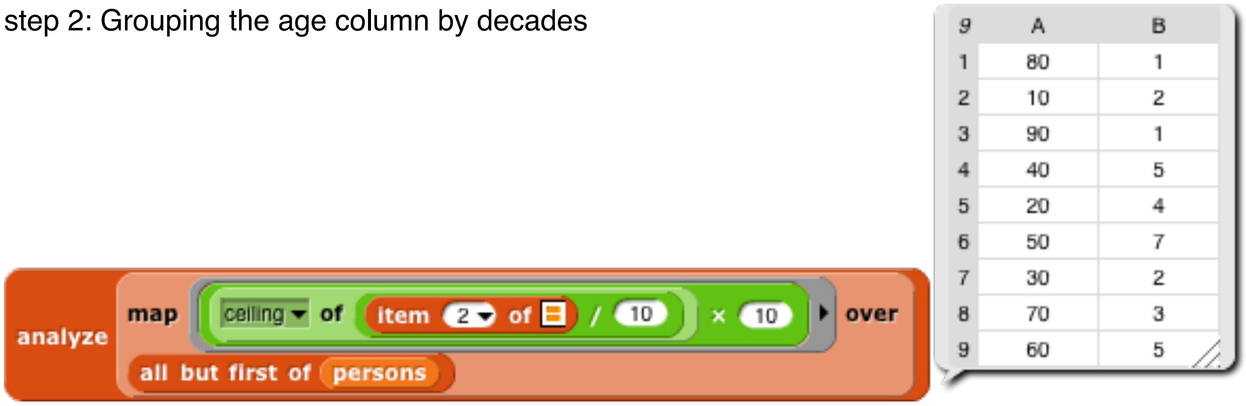
step 1: Looking at the exact ages produces too many keys



```
analyze map (item 2 of ⊟) over all but first of persons
```

27	A	B
1	2	1
2	34	1
3	48	1
4	37	1
5	19	1
6	68	1
7	36	1
8	72	1
9	3	1
10	85	1
11	44	1
12	24	1
13	16	1
14	56	1
15	35	1
16	42	1
17	38	1
18	69	1
19	51	1
20	12	2
21	53	2
22	47	2
23	43	1
24	45	1
25	23	1
26	64	1
27	52	1

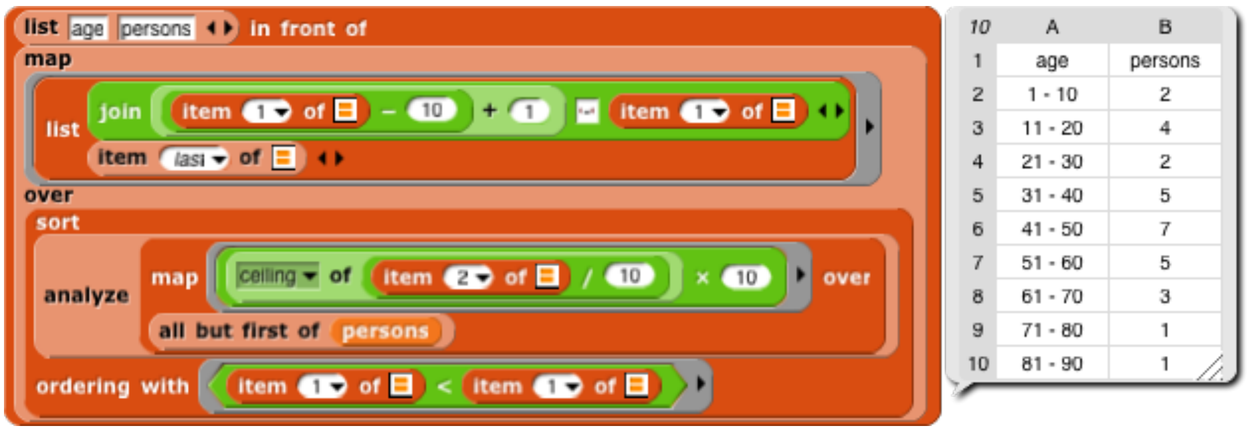
step 2: Grouping the age column by decades



```
analyze map ((ceiling of ((item 2 of ⊟) / 10)) × 10) over
        all but first of persons
```

9	A	B
1	80	1
2	10	2
3	90	1
4	40	5
5	20	4
6	50	7
7	30	2
8	70	3
9	60	5

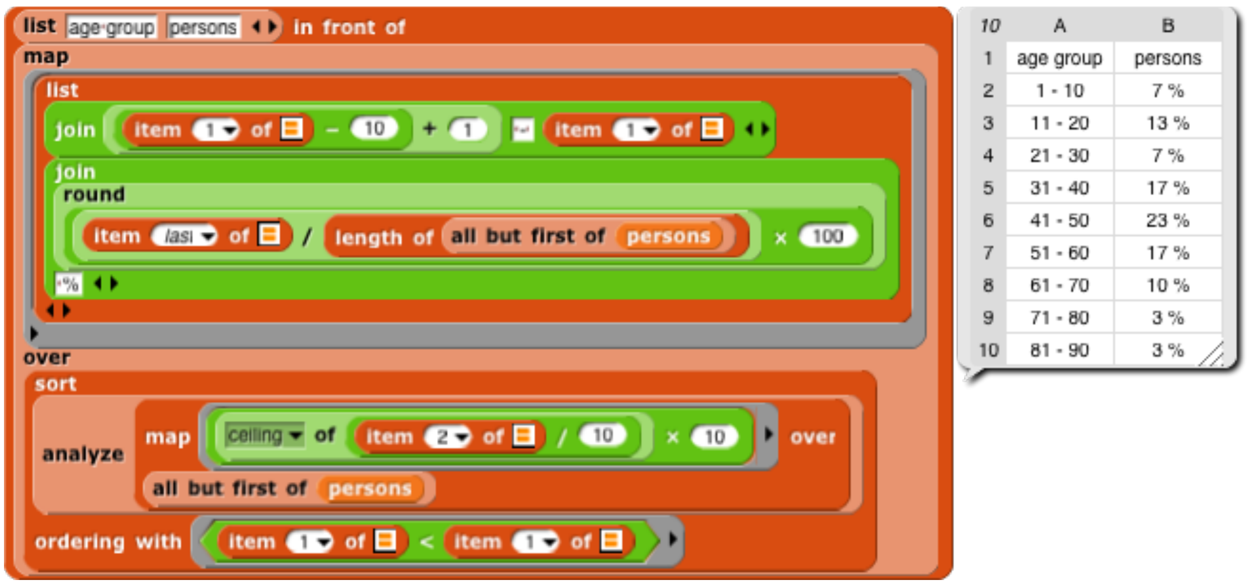
step 3: Transforming ranges for keys and add column labels



```
list age persons ◂▸ in front of
map
    list join item 1 of ☰ − 10 + 1 - item 1 of ☰ ◂▸
         item last of ☰ ◂▸
over
sort
    analyze map ceiling of item 2 of ☰ / 10 × 10 over
                all but first of persons
    ordering with item 1 of ☰ < item 1 of ☰
```

10	A	B
1	age	persons
2	1 - 10	2
3	11 - 20	4
4	21 - 30	2
5	31 - 40	5
6	41 - 50	7
7	51 - 60	5
8	61 - 70	3
9	71 - 80	1
10	81 - 90	1

step 4: Transforming values to percentages

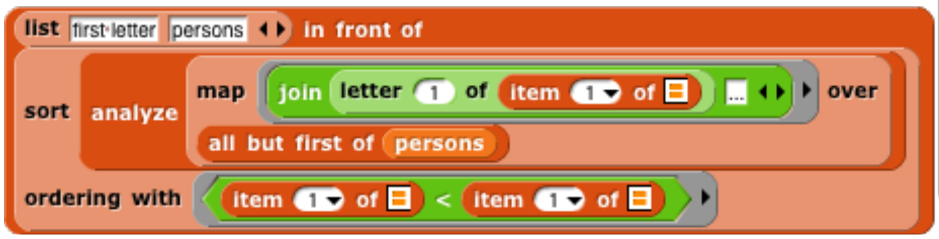


```
list age group persons ◂▸ in front of
map
    list
        join item 1 of ☰ − 10 + 1 - item 1 of ☰ ◂▸
        join
            round
                item last of ☰ / length of all but first of persons × 100
            % ◂▸
        ◂▸
over
sort
    analyze map ceiling of item 2 of ☰ / 10 × 10 over
                all but first of persons
    ordering with item 1 of ☰ < item 1 of ☰
```

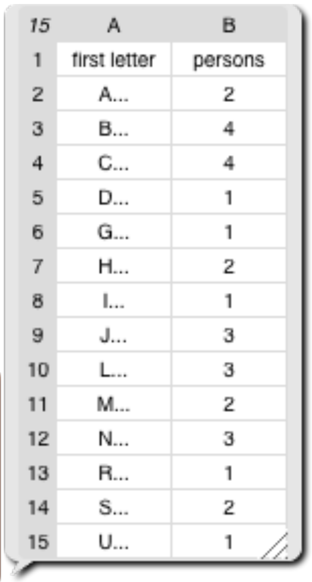
10	A	B
1	age group	persons
2	1 - 10	7 %
3	11 - 20	13 %
4	21 - 30	7 %
5	31 - 40	17 %
6	41 - 50	23 %
7	51 - 60	17 %
8	61 - 70	10 %
9	71 - 80	3 %
10	81 - 90	3 %

Going Meta: Analyzing the Analysis

analyzing first letters:

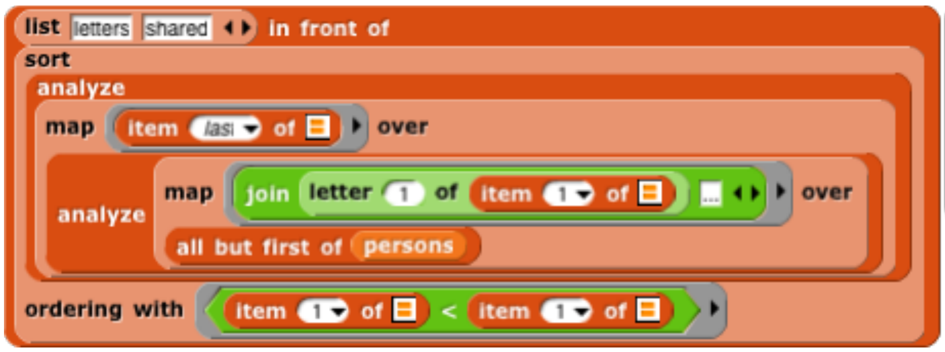


```
list [first letter] [persons] ◂▸ in front of
    sort
        analyze
            map (join (letter 1 of (item 1 of ☰)) ... ◂▸) over
                all but first of persons
    ordering with ((item 1 of ☰) < (item 1 of ☰))
```

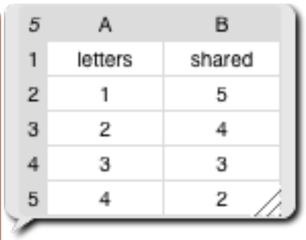


15	A	B
1	first letter	persons
2	A...	2
3	B...	4
4	C...	4
5	D...	1
6	G...	1
7	H...	2
8	I...	1
9	J...	3
10	L...	3
11	M...	2
12	N...	3
13	R...	1
14	S...	2
15	U...	1

Going meta - analyzing the analysis:



```
list [letters] [shared] ◂▸ in front of
    sort
        analyze
            map (item (last) of ☰) over
                analyze
                    map (join (letter 1 of (item 1 of ☰)) ... ◂▸) over
                        all but first of persons
    ordering with ((item 1 of ☰) < (item 1 of ☰))
```

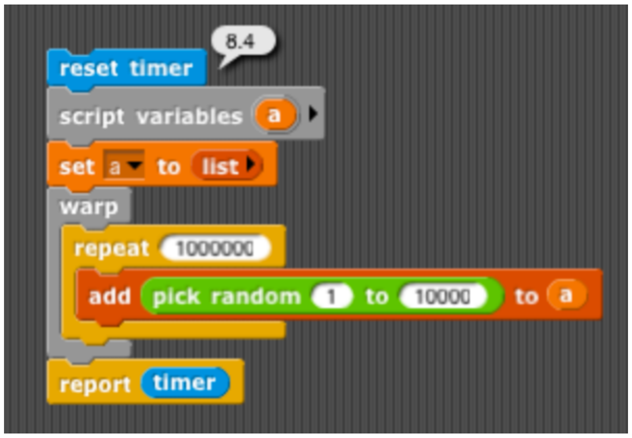


5	A	B
1	letters	shared
2	1	5
3	2	4
4	3	3
5	4	2

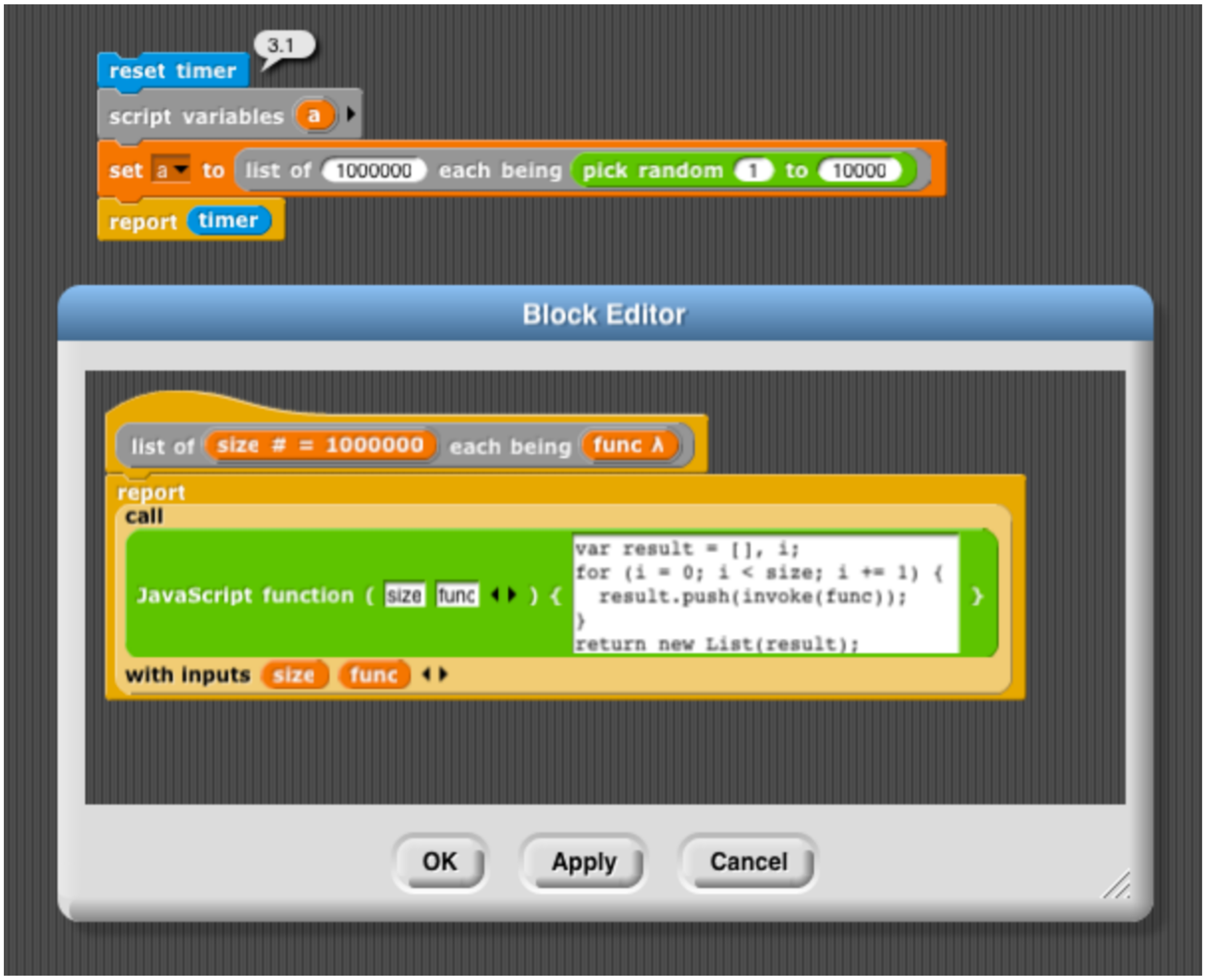
- 5 persons' names have unique first letters.
- 4 pairs of persons share the same first letters in their names.
- 3 letters are shared by three persons' names' first letters.
- 2 groups of 4 persons each share the same first letter in their names.

Fast Blocks

When exploring larger data sets Snap's evaluation speed can be a hindrance, even when WARPing repetitive operations or when using "turbo" mode. For example, creating a list of a million random integers using Snap's standard primitives takes approximately 8.4 seconds on my computer:



This can be alleviated by supplying pseudo-primitives using Snap's JavaScript-function block. a "big idea" in Snap is custom higher order functions. These used to be difficult to write in JavaScript, because JavaScript could not directly evaluate Snap's lambda-blocks (rings) as functions. Since v4.0.4 Snap now provides that ability, enabling significantly faster synchronous custom blocks to be written in JavaScript inside Snap. This way, the same list containing a million random integers can be created in less than half that time:

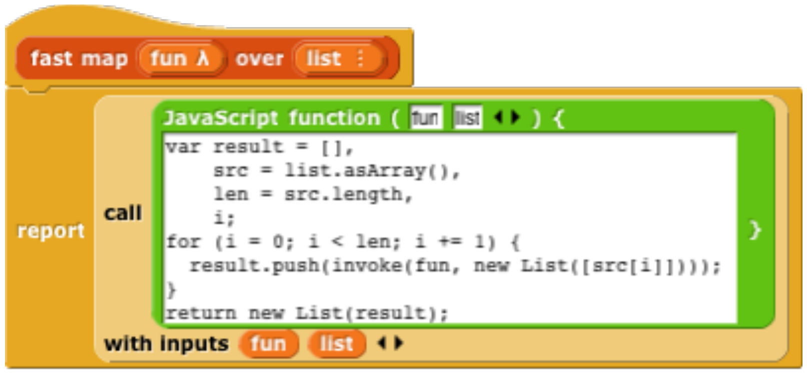

3.1
reset timer
script variables a
set a to list of 1000000 each being pick random 1 to 10000
report timer
Block Editor
list of size # = 1000000 each being func λ
report
call
JavaScript function (size func) {
}
with inputs size func
OK
Apply
Cancel

```
var result = [], i;
for (i = 0; i < size; i += 1) {
  result.push(invoke(func));
}
return new List(result);
```

Carefully providing speed-optimized pseudo-primitives that use Snap's new invoke() JavaScript function for higher-order procedures makes exploring bigger data sets more immediate and enjoyable.

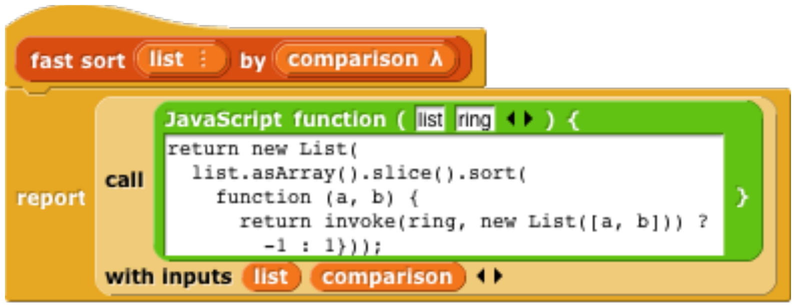
Here are two examples for general purpose speed-optimized higher-order Snap blocks, MAP and SORT, both utilizing this method:

Fast MAP


fast map fun λ over list ⋮
report
call
JavaScript function (fun list ◂▸) {
}
with inputs fun list ◂▸

```
var result = [],
    src = list.asArray(),
    len = src.length,
    i;
for (i = 0; i < len; i += 1) {
  result.push(invoke(fun, new List([src[i]])));
}
return new List(result);
```

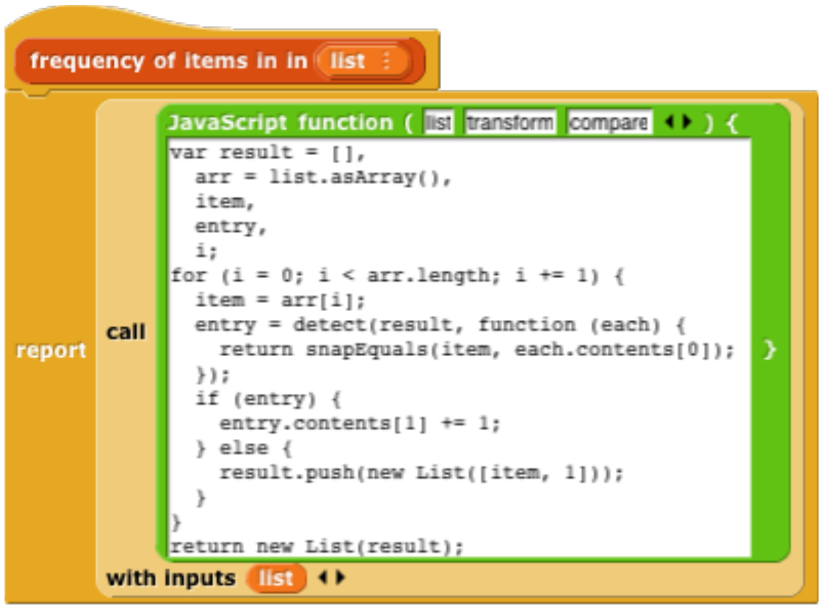
Fast SORT


fast sort list ⋮ by comparison λ
report
call
JavaScript function (list ring ◂▸) {
}
with inputs list comparison ◂▸

```
return new List(
  list.asArray().slice().sort(
    function (a, b) {
      return invoke(ring, new List([a, b])) ?
        -1 : 1}));
```

As you can see in the textual JavaScript code, you can simply use "invoke()" to call a Snap-Ring. This way, Snap blocks - sorta - become first-class JavaScript citizens, much as JavaScript functions can be invoked within Snap using the RUN and CALL blocks and thus have become first-class Snap objects.

Fast ANALYZE


frequency of items in in list
report
call
JavaScript function (list transform compare) {
}
with inputs list

```
var result = [],
  arr = list.asArray(),
  item,
  entry,
  i;
for (i = 0; i < arr.length; i += 1) {
  item = arr[i];
  entry = detect(result, function (each) {
    return snapEquals(item, each.contents[0]);
  });
  if (entry) {
    entry.contents[1] += 1;
  } else {
    result.push(new List([item, 1]));
  }
}
return new List(result);
```

Codification

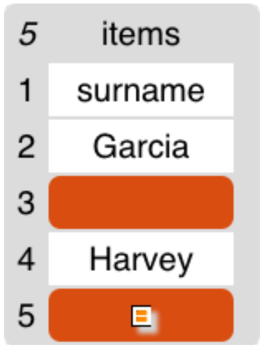
Running an interpreter of an interactive visual programming language inside a browser tab is bound to hit resource and performance limits rather sooner than later. For “bigger” data sets a more promising strategy might be to store them in a server-hosted data base and to use Snap as a client. Snap’s codification feature can be leveraged to transcompile blocks into SQL queries than can be sent to the server hosting the (possibly remote) data base using Snap’s HTTP block. This way, only sample data or smaller sized query results would have to be processed inside the Snap client - and inspected with Snap’s table view widget.

Final Quiz :)

Explain this table:

How was it created?

Enjoy!
-Jens



5	items
1	surname
2	Garcia
3	🟧
4	Harvey
5	🟧