The Flex Array User's Guide

Flexible, Easy Multidimensional Array Processing

by Peter Camilleri

Last Update: February 20, 2016
Covering flex_array version 0.3.2

Table of Contents

The MIT License (MIT).

Introduction

Welcome to the flex array user's guide. This document will explore how to use the flex array, look at some example scenarios, and give a brief overview of the internal design and implementation of the library.

Philosophy

The flex array gem is an amalgam of ideas and approaches from diverse languages including Algol, Pascal, APL, and of course, Ruby. It attempts to meld these concepts into a whole that is compatible with the spirit of Ruby and the goals of performance and ease of learning and use.

The flex array continues the philosophy of enforcing error checking through exceptions and the goal of detecting problems as early as possible with error messages that are as clear as possible.

As always, suggestions and words of encouragement are most appreciated. Those who would offer constructive criticism are also welcome, just don't expect any cake.[1]

Dependencies

The flex_array gem depends on the in_array gem at run time. In development, it utilizes the rake, reek, and mini test utilities.

FlexArray also mixes in the built-in Enumerable module.

1 As always, The Cake is a Lie.

Flex Array Basics

The flex array gem is supported by a large number of methods, but in order for these methods to make much sense, two key concepts are required. These are the array specification, which describes the parameters required to construct a flex array, and a selection, which describes an element, or a portion of, or all of the data within a flex array.

Array Specs

An array specification is used to describe the parameters of a flex array under construction. This specification consists of an array of dimension specifications. Each of these describes the allowable indexes for that dimension. Specifications may be:

Specification	Description	Example
A positive integer	A positive integer, n, defines a dimension with a range of 0…n. This corresponds closely to the array semantics of standard Ruby arrays.	0 2 42
A positive range	A range of positive integers defines a dimension with the same range.	0..2 0...2 1..10

Some examples of array specifications are:

```
[4]        #An array of 4 elements, indexes 0..3
[4,4]      #An array of 16 elements, indexes 0..3, 0..3
[1..10]    #An array of 10 elements, indexes 1..10
[4, 1..10] #An array of 40 elements, indexes 0..3, 1..10
```

Special case for vectors:

For the special case where the flex array being created is a one dimensional, or a vector, the specification does not need to be contained in an array. It may be simply be passed in as itself, without an enclosing array. While this yields a slight savings of notation, it is inconsistent and leads to confusion and errors. Thus this feature should be considered *deprecated a*nd array specifications should always be packaged in an array.

Selections

A selection is used to access parts of or all of a flex array. The flexibility of the selection process is a large part of the flexibility in the name of flex arrays. The following table shows how cells may be selected. The examples given in the table are for an array with a specification of 1..10.

Selection	Description	Example
A positive number	An integer that is greater or equal to zero, selects an array cell relative to the lower limit of the dimension. The value must be constrained by the allowable values or an error will occur.	1 *Selects the first cell.*
A negative number	An integer that is less than to zero, selects an array cell relative to the upper limit of the dimension. The value must be constrained by the allowable values or an error will occur.	-1 *Selects the last cell.*
A range	A range of values greater or equal to zero, selects the array cells from the beginning of the range through to the end.	2..9 *Selects the second through ninth cells.*
	Like integers, ranges use negative values to index relative to the end of the array dimension.	-2..-1 *Selects the second last and last cells.*
	Positive and negative values may be mixed so long as the index represented by the end value is greater than the beginning value.	1..-2 *Selects the second through second last cells.*
An array of integers	An array of integers allows the selection of individual index values in any order. Negative values are relative to the end of the dimension. Values may be omitted or repeated.	[0,-1,1,-2] *Selects the indicated values.*
:all	This symbol selects all of the cells controlled by this dimension.	:all *Selects all 10 cells.*

Special case for selecting the entire array:

A single all selector, that is [:all], will specify all elements of the array, regardless of the number of dimensions in the specification. Many operations are optimized to run faster for this special case.

Flex Array Operations

Creating Flex Arrays:

There are many ways available to create flex arrays. Most of these involve methods of the FlexArray class.

new	FlexArray.new([*array_spec*], *obj=nil*) → *a_flex_array*
	FlexArray.new([*array_spec*]) {|i| … } → *a_flex_array*

Create a new FlexArray instance of the given specification and initialize it to nil, or a value, or the values returned by an initialization block. The block is passed a parameter that consists of an array with the subscripts of the current array element. Note that the same array is reused for each element so if it needs to be stored, a dup or clone of it should be used instead.

See *Array Specs* above for more information on array specs.

```
a1 = FlexArray.new([10,10])                  #Initialized to nil
a2 = FlexArray.new([10,10], 0)               #Initialized to 0
a3 = FlexArray.new([1..12,1..12]) {|i| i[0]*i[1]} #Multiplication table.
```

new_from	FlexArray.new_from([*array_spec*], *flex_array*) → *a_flex_array*
	FlexArray.new_from([*array_spec*], *array*) → *a_flex_array*

Create a new FlexArray instance of the given specification and initialize it with the data of the given flex or standard array. If the created array has fewer elements than the source array, the additional elements are ignored. If the created array has more elements than the source array, the data source cycles back to the start and source elements are repeated as needed.

See *Array Specs* above for more information on array specs.

```
a1 = FlexArray.new_from([10,10], a0)      #Fills a1 with data from a0
a2 = FlexArray.new_from([10,10], [1,2,3]) #Fills a2 with 1,2,3,1,2,3...
```

Note: The following are equivalent:

```
a_flex_array.reshape([array_spec])
FlexArray.new_from([array_spec], a_flex_array)
```

new_from_array	FlexArray.new_from_array(*flex_array*) → *a_flex_array*
	FlexArray.new_from_array(*array*) → *a_flex_array*

These methods are used to create flex arrays that are the same shape as the given flex array or standard array. While the flex array itself and its specifications are distinct, the underlying data is a reference to the underlying data of the source. This has a number of ramifications:

- The underlying data is not copied. This is much quicker.
- No memory is allocated for the underlying data.
- Since it is a reference, changes to the underlying data affect the source array's data.

Note: When the source is a standard array, the array specification of the flex array created is [0...n] where n is the number of elements in the standard array.

```
a0 = FlexArray.new([10,10]) { |i| i[0]==i[1] ? 1 : 0 }
a1 = FlexArray.new_from_array(a0)
a2 = FlexArray.new_from_array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

new_from_selection	FlexArray.new([*array_spec*], *flex_array, [selection]*) → *a_flex_array*

Create a new FlexArray instance of the given specification and initialize it with the data selected from the given flex array. If the created array has fewer elements than the source data selection, the additional elements are ignored. If the created array has more elements than the source data selection, the data source cycles back to the start and source elements are repeated as needed.

```
tt = FlexArray.new([1..12,1..12]) {|i| i[0]*i[1]}
t5 = FlexArray.new_from_selection([12], tt, [5, :all])
```

See *Array Specs* above for more information on array specs.

See *Selections* above for more information on array selections.

to_flex_array	*flex_array*.to_flex_array → *a_flex_array*
	array.to_flex_array → *a_flex_array*

Return the source flex or standard array as a flex array. The returned flex array is always an alias for the source.

Note: When the source is a standard array, the array specification of the flex array created is [0...n] where n is the number of elements in the standard array.

```
a0 = FlexArray.new([10,10]) { |i| i[0]==i[1] ? 1 : 0 }
a1 = a0.to_flex_array
a2 = [1,2,3].to_flex_array
```

dup	*flex_array*.dup → *a_flex_array*

Create a duplicate of the source flex array. The duplicate has distinct array specifications and underlying data. However, the data themselves remain shared, so if those data are mutable, changes in the source/destination are reflected in the other.

```
a1 = a2.dup
```

Note: This method overrides the system dup method which is aliased to shallow_dup.

Reshaping Flex Arrays:

Another flexibility in flex arrays is the ability to change the shape or respecify the dimensions of a flex array. The methods in this section deal with this reshaping.

reshape	*flex_array*.reshape([*array_spec*]) → *a_flex_array*

Create a new array that is a reshaped version of the source flex array. The source and new arrays are distinct and do not share references. However, the data themselves remain shared, so if those data are mutable, changes in the source/destination are reflected in the other. If the created array has fewer elements than the source data, the additional elements are ignored. If the created array has more elements than the source data, the data source cycles back to the start and source elements are repeated as needed.

```
a0 = FlexArray.new([9,9]) { |i| i[0]==i[1] ? 1 : 0 }
a1 = a0.rehape([3,3,3,3])
```

Note: The following are equivalent:

```
a_flex_array.reshape([array_spec])
FlexArray.new_from([array_spec], a_flex_array)
```

reshape! *flex_array*.reshape!([*array_spec*]) → *flex_array*

Reshape this flex array “in place”. In all other aspects, this method is the same as the simple reshape method.

```
a1 = FlexArray.new([9,9]) { |i| i[0]==i[1] ? 1 : 0 }
a1.rehape!([3,3,3,3])
```

Note: The following are all mostly equivalent:

```
a_flex_array.reshape!([array_spec])
a_flex_array = a_flex_array.reshape([array_spec])
a_flex_array = FlexArray.new_from([array_spec], a_flex_array)
```

transpose *flex_array*.transpose(int, int) → *flex_array*

Create a flex array, referencing this one, with the roles of the specified dimensions, transposed or exchanged. Note that the special :all optimizations are disabled for the transposed version of the array.

```
a1 = FlexArray.new([3,3]) { |i| i[0]*3 + i[1]}
#yields: 0 1 2
#        3 4 5
#        6 7 8
a2 = a1.transpose(0,1)
#yields: 0 3 6
#        1 4 7
#        2 5 8
```

transpose! *flex_array*.transpose!(int, int) → *flex_array*

Transpose this flex array “in place” In all other aspects, this method is the same as the simple transpose method. Note that the special :all optimizations are disabled for the transposed version of the array.

```
a1 = FlexArray.new([2,2]) { |i| i[0]*2 + i[1]}
#yields: 0 1
#        2 3
a1.transpose!(0,1)
#yields: 0 2
#        1 3
```

to_a	*flex_array*.to_a → *an_array*

Convert the flex array to a simple array. Any arrays contained in the underlying data array are not affected.

```
a1 = FlexArray.new([9,9]) { |i| i[0]==i[1] ? 1 : 0 }
my_array = a1.to_a
```

Accessing/Updating Flex Arrays:

[]	*flex_array*[*selection] → *an_object* or *an_array*

The element reference operator is used to read one or more elements in the flex array. If the selection resolves to a single element, then that element is returned. If the selection references multiple elements, then a standard array containing those elements is returned.

Notes:

1. See *Selections* above for more information on array selections.
2. An IndexError exception is raised on an out of range subscript value.
3. The selection parameter should only be enclosed in a single set of brackets, and not two. This is because the selection is passed as a splat parameter. So for example, myarray[3,4] is correct while myarray[[3,4]] means something else entirely.

```
a1 = FlexArray.new([2,4]) { |i| i[0]*4 + i[1]}
# a1 equals 0 1 2 3
#           4 5 6 7
a1[0,0]         # 0
a1[0, 0..2]     # [0,1,2]
a1[-1, 0]       # 4
a1[0, -2..-1]  # [2,3]
a1[0, [1, -2]] # [1,2]
a1[:all, 0]     # [0,4]
a1[0, :all]     # [0,1,2,3]
a1[:all, 1..2] # [1,2,5,6]
```

[]=	*flex_array*[*selection] = *an_object* → *an_object*
	flex_array[*selection] = *an_array* → *an_array*
	flex_array[*selection] = *a_flex_array* → *a_flex_array*

The element assignment operator is used to write one or more elements in the flex array. If the selection resolves to a single element, then that element is updated. If the selection references multiple elements, then each of elements is updated.

Notes:

1. See *Selections* above for more information on array selections.
2. An IndexError exception is raised on an out of range subscript value.
3. The selection parameter should only be enclosed in a single set of brackets, and not two. This is because the selection is passed as a splat parameter. So for example, myarray[3,4] = 'x' is correct while myarray[[3,4]] = 'x' means something else entirely.

```
a1 = FlexArray.new([2,4]) { |i| i[0]*4 + i[1]}
# a1 equals 0 1 2 3
#           4 5 6 7
a1[0,0] = 8
# a1 equals 8 1 2 3
#           4 5 6 7
a1[0, 0..2] = 9
# a1 equals 9 9 9 3
#           4 5 6 7
a1[-1, 0] = 0
# a1 equals 9 9 9 0
#           4 5 6 7
a1[0, -2..-1] = 1
# a1 equals 9 9 1 1
#           4 5 6 7
a1[0, [1, -2]] = 0
# a1 equals 9 0 0 1
#           4 5 6 7
a1[:all, 0] = 3
# a1 equals 3 0 0 1
#           3 5 6 7
a1[0, :all] = 4
# a1 equals 4 4 4 4
#           3 5 6 7
a1[:all, 1..2] = 9
# a1 equals 4 9 9 4
#           3 9 9 7
```

Enumerable Flex Arrays:

To support the Enumerable mixin, the flex array implements the each method. For greater compatibility with standard arrays, it also adds the collect! method.

The flex array replaces and extends several methods in the Enumerable module to better support their multidimensional nature. In particular, since indexes into flex arrays are themselves arrays, new versions of each_with_index and find_index where mandatory. Other methods such as collect and cycle need new implementations to handle flex array side effects of features like transposing and appending.

In addition, most methods are available in an enhanced form that permits operations to be conducted on a select portion of the flex array. For example, the each method is also available as the select_each method which has an additional argument of a selection criteria.

See *Selections* above for more information on array selections.

flatten_collect	*flex_array*.flatten_collect {|item| … } → *an_array*
	flex_array.flatten_collect → *an_enumerator*
select_flatten_collect	*flex_array*.select_flatten_collect([*selection*]) {|item| … } → *an_array*
	flex_array.select_flatten_collect([*selection*]) → *an_enumerator*

Returns a new standard array containing the results of running block once for every element in the flex array. Optionally, a selection may be provided to perform the collect operation on a sub-set of the flex array. Returns an Enumerator object if no block is given.

See *Selections* above for more information on array selections.

collect	*flex_array*.collect {|item| … } → *a_flex_array*
select_collect	*flex_array*.selectcollect ([*selection*]) {|item| … } → *a_flex_array*

Returns a new flex array containing the results of running block once for every element in the flex array. Optionally, a selection may be provided to perform the select_collect operation on a sub-set of the flex array. Note: The block parameter is mandatory.

See *Selections* above for more information on array selections.

collect!	*flex_array*.collect! {|item| … } → *a_flex_array*
select_collect!	*flex_array*.selectcollect! ([*selection*]) {|item| … } → *a_flex_array*

Invokes block once for each element of the flex array, replacing the element with the value returned by the block. Optionally, a selection may be provided to perform the select_collect! operation on a sub-set of the flex array. Note: The block parameter is mandatory.

See *Selections* above for more information on array selections.

cycle	*flex_array*.cycle {|item| … } → nil
	flex_array.cycle(count) {|item| … } → nil
	flex_array.cycle → *an_enumerator*
	flex_array.cycle(count) → *an_enumerator*
select_cycle	*flex_array*.select_cycle([*selection*]) {|item| … } → *nil*
	flex_array.select_cycle([*selection*], count) {|item| … } → *nil*
	flex_array.select_cycle([*selection*]) → *an_enumerator*
	flex_array.select_cycle([*selection*], count) → *an_enumerator*

This method is used to cycle through the flex array or the selected portion of it forever or the specified number of times. It is similar to the cycle method of standard arrays.

See *Selections* above for more information on array selections.

each	*flex_array*.each {|item| … } → *a_flex_array*
	flex_array.each → *an_enumerator*
select_each	*flex_array*.select_each([*selection*]) {|item| … } → *a_flex_array*
	flex_array.select_each([*selection*]) → *an_enumerator*

Iterate through each element or each selected element of the flex array. Similar to each as provided by standard arrays.

See *Selections* above for more information on array selections.

each_with_index	*flex_array*.each_with_index {|item,index| … } → a_*flex_array* *flex_array*.each → *an_enumerator*
select_each_with_index	*flex_array*.select_each_with_index([*selection*]) {|item, index| … } → a_*flex_array* *flex_array*.select_each_with_index([*selection*]) → *an_enumerator*

Iterate through each element or each selected element of the flex array. Similar to each_with_index as provided by standard arrays.

See *Selections* above for more information on array selections.

_each_raw	*flex_array*._each_raw {|data, index, posn| … } → a_*flex_array* *flex_array*._each_raw → *an_enumerator*
_select_each_raw	*flex_array*._select_each_raw([*selection*]) {|data, index, posn| … } → a_*flex_array* *flex_array*._select_each_raw([*selection*]) → *an_enumerator*

This is a specialized low level version of each that gives access to the low level data array. The primary use of this method is to emulate other methods. The block parameter is passed three arguments:

1. The low level standard data array.
2. An array containing the index of the current data cell.
3. An integer that is the index of the current data cell in the low level standard array.

See *Selections* above for more information on array selections.

find_index	*flex_array.find_index(obj) → an_array or nil*
	flex_array.find_index {|obj| ... } → an_array or nil
	flex_array.find_index → an_enumerator
select_find_index	*flex_array.select_find_index(*[*selection*]*, obj) → an_array or nil*
	flex_array.select_find_index([*selection*]*) {|obj| ... } → an_array or nil*
	flex_array.select_find_index([*selection*]*) → an_enumerator*

Returns the index of the first object in flex array that is == to obj or for which the block returns a value of true. In the standard implementation of find_index, an integer is returned for the index value. Since multiple dimensions are supported, in the flex array implementation, the index is contained in an array with one element for each dimension in the flex array. If a match is not found nil is returned.

See *Selections* above for more information on array selections.

find_indexes	*flex_array.find_indexes(obj) → an_array*
	flex_array.find_indexes {|obj| ... } → an_array
	flex_array.find_indexes → an_enumerator
select_find_indexes	*flex_array.select_find_indexes(*[*selection*]*, obj) → an_array*
	flex_array.select_find_indexes([*selection*]*) {|obj| ... } → an_array*
	flex_array.select_find_indexes([*selection*]*) → an_enumerator*

Returns the indexes of all the objects in flex array that are == to obj or for which the block returns a value of true. These indexes are contained in a an array, and each element of that array is an array with one element for each dimension in the flex array. If a match is not found an empty array is returned.

Appending to Flex Arrays:

<<	*flex_array* << *an_object* → *flex_array*
	flex_array << *an_array* → *flex_array*
	flex_array << *a_flex_array* → *flex_array*

Like standard arrays, flex arrays support appending additional data onto the “end”. The definition of the “end” of the array is a bit more complex when dealing with flex arrays. Given an N dimensional flex array, the end of the array is represented by the cells starting with an index one beyond the range of the first index and the data appended should match exactly all of the other dimensions of the receiving array. This is expressed thus:

Given a flex array with dimensions

```
[a,b,c, … ]
```

candidate arrays for appending must have dimensions

```
[b,c, … ] or [j,b,c, … ]
```

Note that non-array objects are treated as if they were arrays of specification [1] and standard arrays are treated as arrays if specification [length]. Consider the following examples of the append method in action:

```
a1 = FlexArray.new([1], 0)    #[0]
a1 << 1                       #[0, 1]
a1 << [2, 3, 4]               #[0, 1, 2, 3, 4]

a2 = FlexArray([3,3]) {|i| i[0]*3 + i[1] }
# [ 0, 1, 2
#   3, 4, 5
#   6, 7, 8]
a2 << [9, 10, 11]
# [ 0, 1, 2
#   3, 4, 5
#   6, 7, 8,
#   9,10,11]

a3 = FlexArray([0,3])         #[]
a3 << [1,2,3]                 #[1,2,3]
a3 << [4,5,6]                 #[1,2,3
                              # 4,5,6]
```

Other Operations:

<=>	*a_flex_array* <=> *an_object* → *+1, 0, -1, or nil*

The flex array version of the standard compariositality operator.

==	*a_flex_array* == *an_object* → *a_boolean*

The flex array version of the standard equality operator.

compatible?	*a_flex_array*.compatible?(an_object) → *a_boolean*

Is this flex array compatible with an_object? To be compatible, an_object must be a flex array with the same number and range of array specifications.

empty?	*a_flex_array*.empty? → *a_boolean*

Is this flex array devoid of data elements? This method returns true if there are no data cells allocated.

limits	*a_flex_array*.limits → *an_array*

Returns an array of the limits of indexes of each dimension. The number of elements in this resulting array is the same as the number of dimensions in the flex array.

version	FlexArray.version → *a_string* *a_flex_array*.version → *a_string*

Returns a string with the version of the flex array gem. Note that both the FlexArray class or any flex array object may be queried.

```
puts FlexArray.version  #Currently displays the string "0.3.0".
```

FlexArray Properties

array_data [r/w]

This is the low level, one dimensional backing storage for the flex array. It is exposed as an accessor primarily for speed and testing purposes.

array_specs [r/w]

This is the specification holder for the flex array. It is exposed as an accessor primarily for speed and testing purposes.

transposed [r]

Is this array currently transposed? A Boolean value. Note that append operations (<<) are not permitted on flex arrays that are transposed.

Working with the Enumerable mixin

The FlexArray class mixes in the Enumerable module. While that module is generally outside the scope of this user's guide. this section reviews some aspects that apply to flex arrays. In particular, many methods of Enumerable benefit from being cascaded on other methods. In general the code pattern used is as follows:

```
select_each([selection]).method
```

where method represents the cascaded method from the enumerable module. Some of these methods are:

Method	Cascaded method.
all?	select_each([selection]).all?
any?	select_each([selection]).any?
chunk	select_each([selection]).chunk
count	select_each([selection]).count
detect	select_each([selection]).detect
find_all	select_each([selection]).find_all
first	select_each([selection]).first
flat_map	select_each([selection]).flat_map
grep	select_each([selection]).grep

Method	Cascaded method.
group_by	select_each([selection]).group_by
include?	select_each([selection]).include?
inject	select_each([selection]).inject
max	select_each([selection]).max
max_by	select_each([selection]).max_by
min	select_each([selection]).min
min_by	select_each([selection]).min_by
minmax	select_each([selection]).minmax
minmax_by	select_each([selection]).minmax_by
none?	select_each([selection]).none?
one?	select_each([selection]).one?
partition	select_each([selection]).partition
reject	select_each([selection]).reject
reverse_each	select_each([selection]).reverse_each
sort	select_each([selection]).sort
zip	select_each([selection]).zip

See Ruby documentation for further information on the preceding Enumerable methods.

See *Selections* above for more information on array selections.

Working with the InArrayAlready mixin

The (very tiny) InArrayAlready module is provided via the (terribly small) in_array gem that is used by the flex_array gem. The in_array gem is a very tiny gem that provides for an simple, easy method of ensuring that data is contained in an array with the in_array method.

in_array	*flex_array*.in_array → *flex_array*

Since data in a FlexArray is already in an array, the in_array method simply returns self when sent to a flex array.

A Simple Example

Consider the case of creating a flex array for the purpose of holding a chess board. The code to accomplish this task might look like:

```
#Create an empty chess board.
chess_board = FlexArray.new([8, 8]) do |index|
  color = (index[0] ^ index[1]).even? :black : :white
  ChessPosition.new(color, :empty)
end
```

Potential Pitfalls

Shared References: Flex array shares this pitfall with standard Ruby arrays. When creating the flex array be aware that shared references may produce unexpected results. Consider the following code snippet:

```
my_flex_array = FlexArray.new([5,5], MyThingy.new)
```

In this code, exactly one instance of the MyThingy class is created and the array contains 25 references to that single instance. If instead, 25 MyThingy objects are desired, the following code is required instead:

```
my_flex_array = FlexArray.new([5,5]) { MyThingy.new }
```

This is the same as required by the Ruby built-in Array class. Also like the built-in Array, this problem does not occur with number or symbols or other immutable data, but do not forget that strings are not immutable, and are subject to this issue. One possible solution for strings could be the following:

```
other_array = FlexArray.new([5,5]) { "Brown Cow".dup }
```

Another case is that of the index passed to the new block. It is constantly updated during the creation process. So to create an array filled with its own indexes, consider the following:

```
yet_another = FlexArray.new([5,5]) { |idx| idx.dup }
```

On the other hand, if it was desired to fill an array based, for example, on the products of index values, trickery like `dup` is not needed:

```
times_table = FlexArray.new([1..12, 1..12]) {|idx| idx[0] * idx[1]}
```

Edit History:

PCC – January 26, 2014 – Initial draft of User's Guide started.

PCC – April 2, 2014 – Completed the initial draft for Version 0.3.0.

PCC – April 12, 2014 – Updated with eleventh hour changes to selections and the collect, select_collect, flatten_collect, and select_flatten_collect methods. Also added a brief bit of example code.

PCC – February 20, 2016 – Some formatting fixups.