asarray() — Associative arrays

Description　　Syntax　　Remarks and examples　　Conformability　　Diagnostics　　Also see

Description

asarray() provides one- and multi-dimensional associative arrays, also known as containers, maps, dictionaries, indices, and hash tables.

Also see [M-5] **AssociativeArray()** for a class-based interface into the functions documented here.

Syntax

A =	asarray_create([*keytype*	*declare A*
	[, *keydim*	
	[, *minsize*	
	[, *minratio*	
	[, *maxratio*]]]]])	
	asarray(*A*, *key*, *a*)	*A[key] = a*
a =	asarray(*A*, *key*)	*a = A[key] or a = notfound*
	asarray_remove(*A*, *key*)	*delete A[key] if it exists*
bool =	asarray_contains(*A*, *key*)	*A[key] exists?*
N =	asarray_elements(*A*)	*# of elements in A*
keys =	asarray_keys(*A*)	*all keys in A*
loc =	asarray_first(*A*)	*location of first element or NULL*
loc =	asarray_next(*A*, *loc*)	*location of next element or NULL*
key =	asarray_key(*A*, *loc*)	*key at loc*
a =	asarray_contents(*A*, *loc*)	*contents a at loc*
	asarray_notfound(*A*, *notfound*)	*set notfound value*
notfound =	asarray_notfound(*A*)	*query notfound value*

where

A: Associative array *A[key]*. Created by `asarray_create()` and passed to the other functions. If *A* is declared, it is declared *transmorphic*.

keytype: Element type of keys; `"string"`, `"real"`, `"complex"`, or `"pointer"`. Optional; default `"string"`.

keydim: Dimension of key; $1 \leq keydim \leq 50$. Optional; default 1.

minsize: Initial size of hash table used to speed locating keys in *A*; *real scalar*; $5 \leq minsize \leq 1{,}431{,}655{,}764$. Optional; default 100.

minratio: Fraction filled at which hash table is automatically downsized; *real scalar*; $0 \leq minratio \leq 1$. Optional; default 0.5.

maxratio: Fraction filled at which hash table is automatically upsized; *real scalar*; $1 < maxratio \leq$. (meaning infinity). Optional; default 1.5.

key: Key under which an element is stored in *A*; *string scalar* by default; type and dimension are declared using `asarray_create()`.

a: Element of *A*; *transmorphic*; may be anything of any size; different *A[key]* elements may have different types of contents.

bool: Boolean logic value; *real scalar*; equal to 1 or 0 meaning true or false.

N: Number of elements stored in *A*; *real scalar*; $0 \leq N \leq 2{,}147{,}483{,}647$.

keys: List of all keys that exist in *A*. Each row is a key. Thus *keys* is a *string colvector* if keys are *string scalars*, a *string matrix* if keys are *string vectors*, a *real colvector* if keys are *real scalars*, etc. Note that `rows(`*keys*`)` $= N$.

loc: A location in *A*; *transmorphic*. The first location is returned by `asarray_first()`, subsequent locations by `asarray_next()`. *loc*`==NULL` when there are no more elements.

notfound: Value returned by `asarray(`*A*`, `*key*`)` when *key* does not exist in *A*. *notfound* = `J(0,0,.)` by default.

Remarks and examples

Before writing a program using `asarray()`, you should try it interactively. Remarks are presented under the following headings:

Detailed description

In associative arrays, rather than being dense integers, the indices can be anything, even strings. So you might have *A["Frank Smith"]* equal to something and *A["Mary Jones"]* equal to something else. In Mata, you write that as `asarray(`*A*`, "Frank Smith", `*something*`)` and `asarray(`*A*`, "Mary Jones", `*somethingelse*`)` to define the elements and `asarray(`*A*`, "Frank Smith")` and `asarray(`*A*`, "Mary Jones")` to obtain their values.

A = `asarray_create()` declares (creates) an associative array. The function allows arguments, but they are optional. Without arguments, `asarray_create()` declares an associative array with string scalar keys, corresponding to the *A["Frank Smith"]* and *A["Mary Jones"]* example above.

A = `asarray_create(`*keytype*`,` *keydim*`)` declares an associative array with *keytype* keys each of dimension $1 \times keydim$. `asarray_create("string", 1)` is equivalent to `asarray_create()` without arguments. `asarray_create("string", 2)` declares the keys to be string, as before, but now they are 1×2 rather than 1×1, so array elements would be of the form *A["Frank Smith", "Mary Jones"]*. *A["Mary Jones", "Frank Smith"]* would be a different element. `asarray_create("real", 2)` declares the keys to be real 1×2, which would somewhat correspond to our ordinary idea of a matrix, namely *A[i, j]*. The difference would be that to store, say, *A[100, 980]*, it would not be necessary to store the interior elements, and in addition to storing *A[100, 980]*, we could store *A[3.14159, 2.71828*].

`asarray_create()` has three more optional arguments: *minsize*, *minratio*, and *maxratio*. We recommend that you do not specify them. They are discussed in Setting the efficiency parameters under *Remarks and examples* below.

`asarray(`*A*`,` *key*`,` *a*`)` sets or resets element *A[key]* = *a*. Note that if you declare *key* to be 1×2, you must use the parentheses vector notation to specify key literals, such as `asarray(`*A*`, (100,980), 2.15)`. Alternatively, if `k` = `(100,980)`, then you can omit the parentheses in `asarray(`*A*`, k, 2.15)`.

`asarray(`*A*`,` *key*`)` returns element *A[key]* or it returns *notfound* if the element does not exist. By default, *notfound* is `J(0,0,.)`, but you can change that using `asarray_notfound()`. If you redefined *notfound* to be `0` and defined keys to be real 1×2, you would be on your way to recording sparse matrices efficiently.

`asarray_remove(`*A*`,` *key*`)` removes *A[key]*, or it does nothing if *A[key]* is already undefined.

`asarray_contains(`*A*`,` *key*`)` returns 1 if *A[key]* is defined, and it returns 0 otherwise.

`asarray_elements(`*A*`)` returns the number of elements stored in *A*.

`asarray_keys(`*A*`)` returns a vector or matrix containing all the keys, one to a row. The keys are not in alphabetical or numerical order. If you want them that way, code `sort(asarray_keys(`*A*`), 1)` if your keys are scalar, or in general, code `sort(asarray_keys(`*A*`),` *idx*`)`; see [M-5] **sort( )**.

`asarray_first(`*A*`)` and `asarray_next(`*A*`,` *loc*`)` provide a way of obtaining the names one at a time. Code

```
for (loc=asarray_first(A); loc!=NULL; loc=asarray_next(A, loc)) {
        ...
}
```

`asarray_key(`*A*`,` *loc*`)` and `asarray_contents(`*A*`,` *loc*`)` return the key and contents at *loc*, so the loop becomes

```
for (loc=asarray_first(A); loc!=NULL; loc=asarray_next(A, loc)) {
        ...
        ... asarray_key(A, loc) ...
        ...
        ... asarray_contents(A, loc) ...
        ...
}
```

asarray_notfound(*A*, *notfound*) defines what asarray(*A*, *key*) returns when the element does not exist. By default, *notfound* is J(0,0,.), which is to say, a 0×0 real matrix. You can reset *notfound* at any time. asarray_notfound(*A*) returns the current value of *notfound*.

Example 1: Scalar keys and scalar contents

```
: A = asarray_create()
: asarray(A, "bill", 1.25)
: asarray(A, "mary", 2.75)
: asarray(A, "dan",  1.50)
: asarray(A, "bill")
  1.25
: asarray(A, "mary")
  2.75
: asarray(A, "mary", 3.25)
: asarray(A, "mary")
  3.25
: sum = 0
: for (loc=asarray_first(A); loc!=NULL; loc=asarray_next(A, loc)) {
>         sum = sum + asarray_contents(A, loc)
> }
: sum
  6
: sum/asarray_elements(A)
  2
```

Example 2: Scalar keys and matrix contents

```
: A = asarray_create()
: asarray(A, "Count", (1,2\3,4))
: asarray(A, "Hilbert", Hilbert(3))
: asarray(A, "Count")
       1   2
    +---------+
  1 |  1   2  |
  2 |  3   4  |
    +---------+

: asarray(A, "Hilbert")
[symmetric]
                  1              2              3
    +--------------------------------------------+
  1 |             1                              |
  2 |            .5    .3333333333               |
  3 |   .3333333333            .25             .2|
    +--------------------------------------------+
```

Example 3: Vector keys and scalar contents; sparse matrix

```
: A = asarray_create("real", 2)
: asarray_notfound(A, 0)
: asarray(A, (   1,     1), 1)
```

```
: asarray(A, (1000,  999), .5)
: asarray(A, (1000, 1000), 1)
: asarray(A, (1000, 1001), .5)
: asarray(A, (1,1))
  1
: asarray(A, (2,2))
  0
: // one way to get the trace:
: trace = 0
: for (i=1; i<=1000; i++) trace = trace + asarray(A, (i,i))
: trace
  2
: // another way to get the trace
: trace = 0
: for (loc=asarray_first(A); loc!=NULL; loc=asarray_next(A, loc)) {
>         index = asarray_key(A, loc)
>         if (index[1]==index[2]) {
>                 trace = trace + asarray_contents(A, loc)
>         }
> }
: trace
  2
```

Setting the efficiency parameters

The syntax asarray_create() is

A = asarray_create(*keytype*, *keydim*, *minsize*, *minratio*, *maxratio*)

All arguments are optional. The first two specify the characteristics of the key and their use has already been illustrated. The last three are efficiency parameters. In most circumstances, we recommend you do not specify them. The default values have been chosen to produce reasonable execution times with reasonable memory consumption.

asarray() works via hash tables. Say we wish to record n entries. The idea is to allocate a hash table of N rows, where N can be less than, equal to, or greater than n. When one needs to find the element corresponding to a key, one calculates a function of the key, called a hash function, that returns an integer h from 1 to N. One first looks in row h. If row h is already in use and the keys are different, we have a collision. In that case, we have to allocate a duplicates list for the hth row and put the duplicate keys and contents into that list. Collisions are bad because, when they occur, asarray() has to allocate a duplicates list, requiring both time and memory, though it does not require much. When fetching results, if row h has a duplicates list, asarray() has to search the list, which it does sequentially, and that takes extra time, too. Hash tables work best when collisions happen rarely.

Obviously, collisions are certain to occur if $N < n$. Note, however, that although performance suffers, the method does not break. A hash table of N can hold any number of entries, even if $N < n$.

Performance depends on details of implementation. We have examined the behavior of asarray() and discovered that collisions rarely occur when $n/N \leq 0.75$. When $n/N = 1.5$, performance suffers, but not by as much as you might expect. Around $n/N = 2$, performance degrades considerably.

When you add or remove an element, `asarray()` examines n/N and considers rebuilding the table with a larger or smaller N; it rebuilds the table when n/N is large to preserve efficiency. It rebuilds the table when n/N is small to conserve memory. Rebuilding the table is a computer-intensive operation, and so should not be performed too often.

In making these decisions, `asarray()` uses three parameters:

> *maxratio*: When $n/N \geq maxratio$, the table is upsized to $N = 1.5n$.
>
> *minratio*: When $n/N \leq minratio/1.5$, the table is downsized to $N = 1.5n$. (For an exception, see *minsize*.)
>
> *minsize*: If the new $N < 1.5minsize$, the table is downsized to $N = 1.5minsize$ if it is not already that size.

The default values of the three parameters are 1.5, 0.5, and 100. You can reset them, though you are unlikely to improve on the default values of *minratio* and *maxratio*.

You can improve on *minsize* when you know the number of elements that will be in the table and that number is greater than 100. For instance, if you know the table will contain at least 1,000 elements, starting *minsize* at 1,000, which implies $N = 1{,}500$, will prevent two rescalings, namely, from 150 to 451, and from 451 to 1,354. This saves a little time.

You can also turn off the resizing features. Setting *minratio* to 0 turns off downsizing. Setting *maxratio* to `.` (missing) turns off upsizing. You might want to turn off both downsizing and upsizing if you set *minsize* sufficiently large for your problem.

We would never recommend turning off upsizing alone, and we seldom would recommend turning off downsizing alone. In a program where it is known that the array will exist for only a short time, however, turning off downsizing can be efficient. In a program where the array might exist for a considerable time, turning off downsizing is dangerous because then the array could only grow (and probably will).

Conformability

`asarray_create(`*keytype*`,` *keydim*`,` *minsize*`,` *minratio*`,` *maxratio*`):`

keytype:	1×1	(optional)
keydim:	1×1	(optional)
minsize:	1×1	(optional)
minratio:	1×1	(optional)
maxratio:	1×1	(optional)
result:	*transmorphic*	

`asarray(`*A*`,` *key*`,` *a*`):`

A:	*transmorphic*
key:	$1 \times$ *keydim*
a:	$r_{key} \times c_{key}$
result:	*void*

`asarray(`*A*`,` *key*`):`

A:	*transmorphic*
key:	$1 \times$ *keydim*
result:	$r_{key} \times c_{key}$

`asarray_remove(`*A*`,` *key*`):`

A:	*transmorphic*
key:	$1 \times$ *keydim*
result:	*void*

`asarray_contains(`*A*`,` *key*`)`, `asarray_elements(`*A*`,` *key*`):`

A:	*transmorphic*
key:	$1 \times$ *keydim*
result:	1×1

`asarray_keys(`*A*`,` *key*`):`

A:	*transmorphic*
key:	$1 \times$ *keydim*
result:	$n \times$ *keydim*

`asarray_first(`*A*`):`

A:	*transmorphic*
result:	*transmorphic*

`asarray_first(`*A*`,` *loc*`):`

A:	*transmorphic*
loc:	*transmorphic*
result:	*transmorphic*

`asarray_key(`*A*`,` *loc*`):`

A:	*transmorphic*
loc:	*transmorphic*
result:	$1 \times$ *keydim*

asarray_contents(*A*, *loc*):

A:	*transmorphic*
loc:	*transmorphic*
result:	$r_{key} \times c_{key}$

asarray_notfound(*A*, *notfound*):

A:	*transmorphic*
notfound:	$r \times c$
result:	*void*

asarray_notfound(*A*):

A:	*transmorphic*
result:	$r \times c$

Diagnostics

None.

Also see

[M-5] **AssociativeArray( )** — Associative arrays (class)

[M-5] **hash1( )** — Jenkins's one-at-a-time hash function

[M-4] **Manipulation** — Matrix manipulation

[M-4] **Programming** — Programming functions

For suggested citations, see the FAQ on citing Stata documentation.