Title

stata.com

AssociativeArray( ) — Associative arrays (class)

Description

`AssociativeArray` provides a class-based interface into the associative arrays provided by `asarray()`; see [M-5] **asarray( )**. The class-based interface provides more tersely named functions, making code written with it easier to read.

Associative arrays are also known as containers, maps, dictionaries, indices, and hash tables.

Syntax

`class AssociativeArray scalar` *A*	*create array with* `string scalar` *keys*
or	
A `= AssociativeArray()`	*create array with* `string scalar` *keys*
A`.reinit(`[*keytype*	`"string", "real", "complex", ...`
[`,` *keydim*	`1` *to* `50`
[`,` *minsize*	*tuning parameter*
[`,` *minratio*	*tuning parameter*
[`,` *maxratio*]]]]]`)`	*tuning parameter*
A`.put(`*key*`,` *val*`)`	*A[key] = val*
val `=` *A*`.get(`*key*`)`	*val = A[key] or val = notfound*
A`.notfound(`*notfound*`)`	*change notfound value*
notfound `=` *A*`.notfound()`	*query notfound value*
A`.remove(`*key*`)`	*delete A[key] if it exists*
bool `=` *A*`.exists(`*key*`)`	*A[key] exists?*
val `=` *A*`.firstval()`	*first val or notfound*
val `=` *A*`.nextval()`	*next val or notfound*
key `=` *A*`.key()`	*key corresponding to val*
val `=` *A*`.val()`	*val yet again*
loc `=` *A*`.firstloc()`	*first location or* `NULL`
loc `=` *A*`.nextloc()`	*next location or* `NULL`
key `=` *A*`.key(`*loc*`)`	*key at location*
val `=` *A*`.val(`*loc*`)`	*val at location*

keys = *A*.keys()	*N* × *keydim matrix of defined keys*
N = *A*.N()	*N, number of defined keys*
A.clear()	*clear array; set N equal to 0*

Remarks and examples

stata.com

Array is computer jargon. A one-dimensional array is a vector with elements *A[i]*. A two-dimensional array is a matrix with elements *A[i, j]*. A three-dimensional array generalizes a matrix to three dimensions with elements *A[i, j, k]*, and so on.

An associative array is an array where the indices are not necessarily integers. Most commonly, the indices are strings, so you might have *A["bill"]*, *A["bill", "bob"]*, *A["bill", "bob", "mary"]*, etc.

Associative arrays created by `AssociativeArray` are one-dimensional, and the keys are string by default. At the same time, the elements may be of any type and even vary element by element. Such an array is created when you code

```
function foo(...)
{
        class AssociativeArray scalar  A
        .
        .
}
```

or, if you are working interactively, you type

```
: A = AssociativeArray()
```

Either way, *A* is now a one-dimensional array with string keys. This is the style of associative array most users will want, but if you want a different style, say, a two-dimensional array with real-number keys, you next use *A*.reinit() to change the style. You code

```
function foo(...)
{
        class AssociativeArray scalar  A
        A.reinit("real", 2)
        .
        .
}
```

or you interactively type

```
: A = AssociativeArray()
: A.reinit("real", 2)
```

This associative array will be like a matrix in that you can store elements such as *A[1, 2]*, *A[2, 1]*:

```
: A.put((1,2),  5)
: A.put((2,1), -2)
```

A.put() is how we define elements or change the contents of elements that are already defined. If we typed

```
: A.put((2,1), -5)
```

A[2, 1] would change from −2 to −5. The first argument of *A*.put() is the key (think indices), and the second argument the value to be assigned. The first argument is enclosed in parentheses because *A* is a two-dimensional array, and thus keys are 1×2.

If we now coded

```
x = A.get((1,2))
y = A.get((2,1))
```

then x would equal 5 and y would equal −5. *A* may seem as if it were a regular matrix, but it is not. One difference is that only *A[1, 2]* and *A[2, 1])* are defined and *A[1, 1]* and *A[2, 2]* are not defined. If we fetched the value of *A[1, 1]* by typing

```
z = A.get((1,1))
```

that would not be an error, but we are in for a surprise, because z will equal J(0,0,.), a real 0×0 matrix. That is AssociativeArray's way of saying that *A[1, 1]* has never been defined. We can change what AssociativeArray returns when an element is not defined. Let's change it to be zero:

```
A.notfound(0)
```

Now if we fetched the value of *A[1, 1]* by typing

```
z = A.get((1,1))
```

z would equal zero. We are on our way to creating sparse matrices! In fact, we have created a sparse matrix. I do not know whether our matrix is 2×2 or 1000×1000 because *A[i, j]* is 0 for all (i, j) not equal to $(1, 2)$ and $(2, 1)$. We will just have to keep track of the overall dimension separately. If I defined

```
A.put((1000,1000), 6)
```

then the sparse matrix would be at least 1000×1000. And our matrix really is sparse and stored efficiently in that *A* contains only three elements.

Creating sparse matrices is one use of associative arrays. The typical use, however, involves the one-dimensional arrays with string keys, and these associative arrays are usually the converse of sparse matrices in that, rather than storing just a few elements, they store lots of elements. One can imagine an associative array

```
: Dictionary = AssociativeArray()
```

in which the elements are string colvectors, with the result:

```
: Dictionary.get("aback")
  [1, 1] = (archaic) toward or situation to the rear.
  [2, 1] = (sailing) with the sail pressed backward against the
           mast by the head.
```

I stored the definition for "aback" by coding

```
:  Dictionary.put("aback",
                  (
                      "(archaic) toward or situation to the rear."
                       \
                      "(sailing) with the sail pressed backward
                       against the mast by the head."
                  ))
```

The great feature of associative arrays is that I could enter definitions for 25,591 other words and still `Dictionary.get()` could find the definition for any of the words, including "wombat", almost instantly. Performance would not depend on entering words alphabetically. They could be defined in any order. A user once complained that we slowed down somewhere between 500,000 and 1,000,000 elements, but that was due to a bug, and we fixed it.

Here is a summary of `AssociativeArray`'s features.

Initialization:

Declare *A* in functions you write,

```
class AssociativeArray scalar  A
```

or if working interactively, create *A* using the creator function:

```
A = AssociativeArray()
```

A is now an associative array indexed by `string scalar` keys. *string scalar* is the default.

Reinitialization:

After initialization, the associative array is ready for use with `string scalar` keys. Use `A.reinit()` if you want to change the type of the keys or set tuning parameters. Keys can be `scalar` or `rowvector` and can be `real`, `complex`, `string`, or even `pointer`.

```
A.reinit([ keytype          "string", "real", "complex", ...
        [, keydim           1 to 50
        [, minsize          tuning parameter
        [, minratio         tuning parameter
        [, maxratio]]]]])   tuning parameter
```

Do not specify tuning parameters. Treat `A.reinit()` as if it allowed only two arguments. You are unlikely to improve over the default values unless you understand how the parameters work. Tuning parameters are described in [M-5] **asarray()**.

Add or replace elements in the array:

Add or replace elements in the array using `A.put()`:

```
A.put(key, val)     A[key] = val
```

Values can be of any element type, `real`, `complex`, `string`, or `pointer`. They can even be structure or class instances. Values can be scalars, vectors, or matrices. Value types do not need to be declared. Value types may even vary from one element to the next.

Retrieve elements from the array:

Retrieve values using `A.get()`:

```
val = A.get(key)     val = A[key] or val = notfound
```

Retrieving a value for a key that has never been defined is not an error. A special value called *notfound* is returned in that case. The default value of *notfound* is `J(0,0,.)`. You can change that:

```
A.notfound(notfound)     change notfound value
```

Users of associative arrays containing numeric values often change *notfound* to zero or missing by coding *A*`.notfound(0)` or *A*`.notfound(.)`.

You can use *A*`.notfound()` without arguments to query the current *notfound* value:

notfound = *A*`.notfound()` *query notfound value*

Delete elements in the array:

Delete elements using *A*`.remove()`:

A`.remove(`*key*`)` *delete A[key] if it exists*

Function *A*`.exists()` will tell you whether an element exists:

bool = *A*`.exists(`*key*`)` *A[key] exists?*

The function returns 1 or 0; 1 means the element exists. You may wonder about the necessity of this function because *A*`.get()` returns *notfound* when an element does not exist. Why are there two ways to do one task? *A*`.exists()` is useful because you could store the *notfound* value in an element. You should not do that, of course.

Iterating through all elements of the array:

There are three ways to iterate through the elements.

Method 1 is

```
for (val=A.firstval(); val!=notfound; val=A.nextval()) {
        key = A.key()       // if you need it
        .
        .
}
```

Inside the loop, `val` contains the element's value. If you need to know the element's key, use *A*`.key()`.

Method 2 is

```
transmorphic  loc
for (loc=A.firstloc(); loc!=NULL; loc=A.nextloc()) {
        val = A.val(loc)
        key = A.key(loc)    // if you need it
        .
        .
}
```

Method 2 allows for recursion. Use method 2 if you call a subroutine inside the loop that itself might iterate through the elements of the array.

Method 3 is an entirely different approach. You fetch the full set of defined keys and loop through them. Function *A*.keys() returns the keys as a matrix. Each row of the matrix is a key.

```
K = A.keys()
for (i=1; i<=length(K); i++) {
        val = A.get(K[i,.])
        .
        .
}
```

The keys returned by *A*.keys() are in no particular order. For some loops, order matters. Use Mata's sort() function to order them. If the keys were of dimension 1 and thus K were $N \times 1$, you could code

```
K = sort(A.keys(), 1)
```

If *A* were $N \times k$, you could code

```
K = sort(A.keys(), (1..k))
```

Vector operator 1..*k* produces the row vector (1, 2, . . . , *k*).

Miscellany:

A.N() returns the number of defined elements in *A*.

A.clear() clears the array *A*. The array's characteristics—key type and dimension, *notfound* value, and tuning parameters—remain unchanged.

Conformability

A.reinit(*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)

A.put(*key*, *a*):

key:	$1 \times$ *keydim*
a:	$r_{key} \times c_{key}$; r_{key} and c_{key} your choice
result:	*void*

A.get(*key*):

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

A.remove(*key*):

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

A.clear():

result: *void*

A.exists(*key*):

key: $1 \times keydim$
result: 1×1

A.N():

result: 1×1

A.keys():

result: $N \times keydim$

A.firstval():

result: *transmorphic*

A.firstloc():

result: *transmorphic*

A.nextval():

result: *transmorphic*

A.nextloc():

result: *transmorphic*

A.key():

result: $1 \times keydim$

A.key(*loc*):

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

A.val():

result: $r_{key} \times c_{key}$

A.val(*loc*):

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

A.notfound(*notfound*):

notfound: $r \times c$; your choice
result: *void*

A.notfound():

result: $r \times c$

Diagnostics

None.

Also see

[M-5] **asarray( )** — Associative arrays

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

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