Chapter 3: The Structure of Data

In this chapter, the concepts of A+ data and the vocabulary used in describing them are discussed first. Then some A+ primitive functions for creating and indexing arrays and for inquiring into their characteristics are introduced. In these sections a number of examples of arrays are given. Finally, certain classes of arrays which are useful in the description of A+ functions are treated.

Concepts and Terminology

The data objects in A+ are *arrays*, which can be visualized as rectilinear arrangements of individual values. An individual value in an array is called an *element*. In the simplest arrays, the elements are either all numbers or all characters. A number or a character is itself an array, of the most elementary kind.

In the rectangular visualization of an array, each set of parallel edges defines a direction. Corresponding to each of these directions is an *axis*. The axes of an array are ordered. In the visualization of an array with three axes, the first axis is directed away from the viewer, the second is directed downward, and the third is directed to the right. A two dimensional display of an array with three axes shows it as a series of planes arranged vertically, representing cross sections perpendicular to the first axis. The term *leading axes* is used for any set composed of all the axes from the first up to some particular axis, inclusive, and *trailing axes* for any set composed of all the axes from some particular axis through the last one.

An array with no axes, necessarily consisting of a single element, is called a *scalar*. All elements of arrays are scalars. Arrays with one axis are called *vectors* or *lists*, or, if character, *strings*. Arrays with two axes are called *matrices*, and sometimes *tables*. A set of elements lying along, i.e., parallel to, the first axis of a matrix is called a *column*, and a set along the second axis a *row*, just the same as in ordinary usage for tables. These terms are also used for elements along the two trailing axes of arrays with more than two axes.

Dimension, Shape, and Rank

The *length of an axis* is the number of elements lying along any one of the edges defining that axis. This length is also called a *dimension*, so an array has as many dimensions as axes. (The word dimension is sometimes used as a synonym for the word axis, but not in this manual.) The vector composed of the lengths of all axes of an array, i.e., the vector of dimensions, is called the *shape* of the array. The ordering of the dimensions in the shape is the same as the ordering of the axes to which they correspond. The total number of elements in an array can be found by multiplying together all the elements of its shape.

An array can be *empty*, that is, it can have no elements at all. An empty array can have any number of axes except zero, which is disallowed, essentially because you can't have an empty container without a container. At least one of the dimensions of an empty array is equal to zero.

The *rank* of an array is the number of its axes, and therefore it is also the number of elements in its shape, i.e., the length of its shape. A scalar has an empty shape—its shape is a vector that has no elements—and its rank is 0. (Incidentally, when all the elements of an empty vector are multiplied together the result is 1, by convention, so that the usual formulation for the number of elements in an array `a`—namely, `«/ a`—works for scalars also.)

Every element of an array can be referenced by a set of coordinates called *indices*, to retrieve the value of the element or to give it a new value. There is one index, or coordinate, for each axis, and A+ defines its value to be an integer between zero and one less than the length of that axis, inclusive. The number of indices of an element in an array, then, equals the rank of the array.

Some computational languages use the word *cell* as a synonym for element, but A+ does not (except in connection with the displays created by s, the screen management system): *cell* is used in connection with the *partitioning* of an array, as defined by a set of leading axes. In practice, multidimensional arrays are commonly viewed as partitioned into collections of lower dimensional arrays. For example, a numeric matrix

containing bond prices may be organized so that the rows are time series of prices for bonds, with one row for each bond of interest, while the columns are collections of prices at particular times. For some calculations the rows would be emphasized, while for others, emphasis would be on the columns. One view represents a partition of the matrix into a collection of row vectors, and the other into column vectors.

A+ emphasizes partitions where the lower dimensional arrays lie along a set of trailing axes. The lower dimensional arrays that comprise such a partition are called cells. The complementary set of leading axes is called the *frame* of the partition that *holds* the cells; the cells are said to be *in their frame*. In the case of the numeric matrix of bond prices, the row vectors are the cells of rank 1, and the first axis is their frame.

Every set of leading axes defines a partition into cells for which it is the frame. The set of all axes is a particular set of leading axes, and therefore defines a partition. Since there are no axes left for the cells, the cells must be the elements of the array; the A+ notion of cell, then, includes the more common one. At the other extreme, the array itself is a cell, i.e., a partition of itself into one subarray. In this case the cell takes all the axes and therefore the frame has no axes.

A cell consists of all those elements that have one particular set of indices for the leading axes that define the partition, and all possible indices for the trailing axes. The entire cell can be selected by specifying only the particular indices for the leading axes. Those leading axes are the frame of the partition, and therefore the frame is, loosely speaking, an array of cells that can be indexed by valid indices of them. A partition creates, then, a view of an array as a frame of cells. There is more about frames and cells, including several examples, later in this chapter. The “Dyadic Operators” chapter, and especially its “Rank Deriving Dyadic” section (page 116), has a further discussion of this subject, with examples.

One partition plays a special role in A+, the one defined by the first axis alone; the cells for this partition are called the *items* of an array. Every array can be regarded as a vector of items, and many A+ functions look at them in just that way. In such a context, a scalar is regarded as having a single item, namely itself.

Type and Nesting

Another characteristic of arrays is type. In a simple array (definition later), all elements have the same type, but a nonsimple array can contain elements of several different types.

The most common simple arrays are numeric and character. Every element of a simple numeric array is a number, and every element of a simple *character* array is a character. Numeric arrays can be of either *integer* or *floating point* type. These two types correspond to whole numbers and fractional (sometimes called decimal) numbers. A+ numeric primitive functions applied to integer arrays may automatically convert their arguments to floating point, like the Matrix Inverse function, or may attempt to produce an integer result, like Add and Subtract. If an overflow occurs during this attempt, the type of the result is changed to floating point.

The type of a simple array may also be *symbol* or *function* if it is nonempty, or *null* if it is empty. A symbol is a character string represented as a single scalar; it is denoted by a backquote followed by the string, as in `` `sym ``. A function expression, e.g., `+.«`, and a function scalar, e.g., `<{-}`, both have type function.

While the elements of arrays are often just individual numbers and characters, an element of an array can be an encapsulated multi-element array. That is, any array can be *enclosed* to become a scalar, and this scalar can be an element of another array. Also, any enclosed array, except a function scalar, can be *disclosed*, in order to work with its contents. (A function scalar is an enclosed function expression. The operator Apply, given a function scalar, produces the underlying function expression.) An array that has an enclosed element other than a function scalar is called *nested*, and one that has no enclosed elements except function scalars is called *simple*. A function scalar is simple, but an enclosed function scalar is nested. Any nested array is necessarily nonempty, being or containing a scalar.

A simple scalar symbol or function scalar can be an element of a nested array. In order for data whose type is character, integer, floating point, function, or null to appear in a nested array, however, it must first be enclosed. Clearly, any nonscalar array must be enclosed before being inserted as an element in another array, since the elements of all arrays must be scalars.

When an array other than a function expression is enclosed, the resulting array is a scalar of type *box*. The type of a nonscalar nested array is the type of its first item. Since a nested array can contain elements whose types are box, symbol, and function, its type can be any one of these three. The disclosure of a box scalar, of course, can yield an array of any type.

Any empty array is simple, because if it were nested, it would contain an enclosed array. An empty array that is reshaped or selected from a character, integer, or floating point array is of the same type. Empty arrays of these three types can also be produced by explicit type transformations from empty arrays of these types. The type of an empty array of symbols, functions, nulls, or boxes is null. The empty vector whose type is null is called Null or the Null; it can be represented as `()`.

There is also a type called `unknown`, to guard against weird cases that might arise. It will not be mentioned further, except in the description of the Type function.

Creating Arrays

A+ provides direct ways to specify constant arrays. A list of numbers separated by blank spaces is one description of a simple constant numeric array. For example, the constant

```
10 2.3e-2 34.156
```

is a floating point array with one axis, of length three. The element at index 0 is 10, at index 1 is .023, and at 2 is 34.156. The expression with `e` means the number on the left times ten to the power shown on the right. If you omit the blanks between numbers—a poor idea indeed, since it would make your code very difficult to read—, A+ will give you a numeric vector, but probably not the one you intended. If a number is being parsed and a character is examined that can't be part of the number, then a new number is started if the character could begin a number. For instance,

`1e-3.5 40.358.62.7` is read by A+ as `0.001 0.5 40.358 0.62 0.7`

Simple symbol vectors can be written similarly, and blanks are not needed. One of length five is

```
‘sym1 ‘sym2 ‘sym3‘sym4‘sym5
```

It is also easy to describe simple constant character vectors. For example,

```
'axrTVw'
```

is a character array with one axis, of length six. The elements at indices 0, 1, 2, 3, 4, and 5 are, respectively, `'a'`, `'x'`, `'r'`, `'T'`, `'V'`, and `'w'`. The empty character vector can be written most easily as `''`—just two quotation marks, with nothing between them.

A nested vector can be described conveniently by a *strand*, a parenthesized expression in which the vector's elements are separated by semicolons. Enclosure of each element is implied by strand notation. For example,

```
(‘sym; +; 1 2 3 4; 1.7 3.14; ’example’;)
```

is a nested vector of length six. The blanks after the semicolons are not required, but usually promote readability. All of its elements except the second are of type box; the second is a simple function scalar. The types (lengths) of its elements when each is disclosed are: symbol (a scalar), function (a scalar), integer (4), floating point (2), character (7), and null (0). The absence of an expression in any position of the strand implies a Null.

Arrays with more than one axis can be formed using the dyadic primitive function called Reshape and denoted by ⍴ (rho). For example, the result of the expression

```
      2 3 'axrTVw'   ⍝ Enter this in an A+ session, and press Enter.
axr                  ⍝ This row and the next display the result.
TVw                  ⍝ Text following "⍝" is a comment.
```

is an array with two axes—a matrix. The left argument of Reshape in this example is a vector, specifying the shape of the result. The index of an element in the matrix is a pair consisting of one index for axis 0, and one for axis 1. For instance, the element `'r'` is indexed by the pair 0, 2.

The monadic primitive function called Interval and denoted by ⍳ (iota) is somewhat like Reshape. It creates arrays of any specified shape whose elements are the integers 0, 1, ... For example,

```
      17      ⍝ Simple vectors are always displayed horizontally.
 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
```

is an array with one axis. Note that this array has 17 elements, and the index of the i-th element is i for every i from 0 to 16.

The Interval primitive can also create arrays with more than one axis. For example:

```
      2 3 5          ⍝ Enter this in an A+ session, and press Enter.
 0  1  2  3  4       ⍝ These seven rows (one blank) show the result,
 5  6  7  8  9       ⍝ which is equal to  2 3 5      2«3«5
10 11 12 13 14       ⍝ (which could be written  2 3 5   2«3«5).
                     ⍝ A blank line separates planes.  If there were a fourth axis,
15 16 17 18 19       ⍝ two blank lines would separate subarrays corresponding to
20 21 22 23 24       ⍝ indices along the first axis, and single blank lines between
25 26 27 28 29       ⍝ subarrays corresponding to indices along the second axis.
```

The empty integer vector is most easily written `⍳0` and the empty floating point vector `0⍴0.` (decimal point). In displays, all empty arrays occupy one (blank) line, except the Null, which occupies no display space at all.

The function Enclose, denoted by `<`, is used to enclose arrays; `<` is used also to indicate enclosure in displays:

```
     < 2 5      ⍝ Much like the previous example, but an enclosed scalar.
<  0 1 2 3 4    ⍝ The < is used to indicate enclosure.
   5 6 7 8 9    ⍝ < is displayed only at the start of each enclosed array.
```

Strand notation can be combined with Enclose:

```
     (1 2 3;<1 2 3;'abc';+;'smbl;)  ⍝ The last element is Null.
<  1 2 3          ⍝ Strand encloses the simple vector.
< <  1 2 3        ⍝ Strand encloses the enclosed vector.
<  abc           ⍝ Enclosed character vector.
<  +              ⍝ Enclosed function expression.
<  `smbl          ⍝ Enclosed symbol.
<                 ⍝ Enclosed Null.
```

Warning! In Version 2, sometimes `<` is displayed to indicate that what follows is a symbol; then no back-quote (`) is shown for the symbol.

Indexing Arrays

A+ provides primitive functions to access the elements of an array. One such function is denoted by the bracket pair `[]` and is called Bracket Indexing. For example, using arrays displayed in the previous section:

```
     'axrTVw'[4]
V
     'axrTVw'[5 0 1]
wax
     (`sym;+;1 2 3 4;1.7 3.14;'example';)[2]
<  1 2 3 4
```

```
    ( 2 3 5)[0;1;3]
8
```

An omitted index implies all permitted indices for that axis, so one can easily obtain a row and a column:

```
    ( 2 3 5)[0;0;]    ⍝ The first row.
0 1 2 3 4
    ( 2 3 5)[0;;4]    ⍝ The fifth column of the first plane of the array;
4 9 14                ⍝ vector result.
```

For a 3-dimensional array, an item is a matrix. In Bracket Indexing, a semicolon may be omitted when all the indices following it are omitted, so one can index an array as if it were a vector containing the array's items:

```
    ( 2 3 5)[1]    ⍝ The second item: any element of it is
15 16 17 18 19     ⍝ ( 2 3 5)[1;j;k] for some j and k.
20 21 22 23 24
25 26 27 28 29
```

More generally, one can index an array of rank r as if it were an (r-n)-rank array (frame) of rank-n cells. Say one has a five dimensional array; one can view it as a three dimensional array of two dimensional cells:

```
    (  4 5 6 2 3)[0;0;0]  ⍝ Any element of the first cell is
0 1 2                     ⍝ (  4 5 6 2 3)[0;0;0;j;k] for some j, k.
3 4 5                     ⍝ The first three indices index the frame.
```

One more example demonstrates the power of working with items, frames, and cells. For this example, a small part of the capability of the primitive function Take ( ) and the primitive operator Rank (@) must be explained. For positive n, the expression `n a` produces the first n items of a. The derived function `@1` applies Take to all cells of rank 1 in its right argument, i.e., to all rows, whose items are elements. Taking a certain number of elements in each row is equivalent to taking a certain number of columns. Thus the following expression takes three rows (items of a matrix) after taking five columns of a five by ten matrix:

```
    3 5 @1  5 10   ⍝  3 (5( @1)  5 10) is equivalent.
 0  1  2  3  4     ⍝       ....
10 11 12 13 14     ⍝       Take 5 columns.
20 21 22 23 24     ⍝ Take 3 rows.
```

Inquiring about Arrays

Shape and Rank

The primitive function denoted by the monadic (i.e., one argument) form of the symbol (rho) is called Shape. It produces the shape vector of its array argument. For example, `2 3` is the vector 2 3, and `'axrTVw'` is the one-element vector whose only element is 6.

The result of `a`, a double application of Shape, is a one-element vector whose value is the rank of a. In particular, the element of the one-element vector `36` or `'X'` is 0; separately entered numbers and characters have no axes, and their rank is therefore 0; they are scalars.

Type and Depth

The Type monadic primitive function (') produces the type of its argument, as a scalar symbol. First, the six types of simple arrays.

```
    ' 2 5
‘int
```

```
    '2.71828 3.14159
‘float
    ''axrTVw'
‘char
    ''‘pp ‘rl
‘sym
    '{+}    ⍝ Parser needs braces as hint that + is an arg.
‘func
    '<{+}   ⍝ A function scalar is also of type ‘func.
‘func
    '()
‘null
```

Next, the three types of nested arrays. The type of a nested array is the type of the first item.

```
    '<2 3 4
‘box
    ''‘rl,(;2.7 3.1) ⍝ Comma concatenates two args.
‘sym
    '(+;)            ⍝ A function scalar.
‘func
```

Last, the four types of empty arrays.

```
    ''’
‘char
    ' 0
‘int
    '0 12 10.1
‘float
    '0 4 (+;-;«; )
‘null
```

The Depth monadic primitive function (⍎) produces the depth of nesting of its argument, as a scalar integer. The depth of a multi-item array is the greatest of the depths of its items. The depth of a function expression is -1, by convention, and the depth of a function scalar, which is an enclosed function expression, is 0.

```
    ⍎ 2 3 4            ⍝ Simple
0
    ⍎<’abc def’        ⍝ Result of Enclose
1
    ⍎(2 3;+;‘a‘b‘c)    ⍝ Enclosure implied by strand
1
    ⍎(<2 3;=;‘a‘b‘c)   ⍝ Strand with enclosed element
2
    ⍎(1 2;(3 4;(5 6;);7);8) ⍝ Strand in strand in strand
3
```

A shorter definition of a *simple* array is any array whose depth does not exceed 0. A *nested* array, which is any array that is not simple, can be defined similarly as one whose depth is at least 1.

Pictorial Representation

A file that shows two dimensional representations of data is distributed with the Version 2 A+ system and resides (after loading) in the `disp` context. Here is a sample of its use:

```
$load disp
disp.disp 2 3 (’ab’;‘abc‘def; 2 4; 1.1 2.2;;«)
```

```
<---------------------------------.
¨.--.       .----------. .--------.¨
¨¨ab¨       ¨'abc 'def ¨ ¨ 0 1 2 3¨¨
¨'--'       '----------.¨ 4 5 6 7¨¨
¨                       '--------'¨
¨.--------. .-.         .-.       ¨
¨¨1.1 2.2 ¨ ¨ ¨         ¨«¨       ¨
¨'--------' '-'         '-'       ¨
'---------------------------------'
```

This file is not distributed with Version 4. A more up-to-date version is available in `/common/a/disp.+`. It is used as follows:

```
     $load /common/a/disp
     disp.disp 2 3 ('ab';'abc'def; 2 4; 1.1 2.2;;«)
+3-----------------------------+
2+2-+       +2-------+ +4------+¨
¨¨ab¨       ¨'abc'def¨ 20 1 2 3¨¨
¨+"-+       +'-------+ ¨4 5 6 7¨¨
¨                     +i------+¨
¨+2-------+ +0        +0+     ¨
¨¨1.1 2.2 ¨ +°        ¨«¨     ¨
¨+f-------+           +¨+     ¨
+ -----------------------------+
```

Consult the ASCII text file `/common/a/disp.doc` for further information.

Subtypes and Supertypes

Slotfillers

A special form of nested array called a *slotfiller* is recognized by certain primitive functions and toolkits. A slotfiller is a two-element vector (`sym;val`). `sym` is a simple vector or scalar of distinct symbols. `val` has the same number of items as `sym` (recall that a scalar has one item). It can be either any nested scalar or any vector each of whose items either has a depth of at least 1 or is a function scalar that is the enclosure of a *defined* function. Thus primitive functions can appear in `val` only when they are enclosed at least twice, i.e., as enclosed function scalars. A slotfiller can be thought of as a dictionary of keys (with no repetitions) and values.

There is a way to test whether a variable or an expression is a slotfiller or not: `_issf x` is 1 if `x` is a slotfiller and 0 if it is not. Cf. the "Is a Slotfiller" section, page 148, in the "System Functions" chapter.

Examples of slotfillers are:

```
     ('small 'medium 'large 'super;(16;32;64;72))
     ('a;<97)
```

and

```
     ('g 'l 'w;(f;g;<{+}))
```

where `f` and `g` are user-defined functions, and `+` is enclosed by `<` and the strand; but not

```
     ('g 'l 'w;(+;-;«))
```

since nonnested primitive functions are prohibited in slotfillers.

Recall that when A+ displays a nested array, it uses an Enclose symbol (`<`) to indicate the beginning of the display of each nested array. It indents subarrays appropriately to show their total depth of nesting. The first sample slotfiller is displayed as:

```
<  ‘small ‘medium ‘large ‘super
< <  16
  <  32
  <  64
  <  72
```

The Pick function (page 82) can extract values from slotfillers:

```
     ‘medium (‘small ‘medium ‘large ‘super;(16;32;64;72))
 32
```

Restricted Whole Numbers

Many functions require as arguments whole numbers that are within the range of integer representation but do not insist that the type of these arguments be integer. They also accept floating point numbers that are tolerably equal to integers (see “Comparison Tolerance”, page 105) and numbers whose absolute value is less than `1e-13`. I.e., they reject floating point numbers that are significantly fractional or that are too large in magnitude to be represented as integer type. Furthermore, they accept empty arrays regardless of type. For convenience in this manual, the term restricted whole number is used for a member of the set consisting of the integers, these floating point near-integers, and all empty arrays.

Since the functions that accept restricted whole number arguments use integers internally, floating point values for these arguments involve a performance penalty, because of the implicit type conversion.

General Types

Many A+ functions and operators take arguments of several types, sometimes with some limitation, and it is convenient to have a terminology dividing A+ data objects into three classes, as they do. In this manual, these classes are called general types. They are:

- character, consisting of simple arrays of characters
- numeric, consisting of simple arrays, unrestricted as to value, of
  - floating point numbers
  - integers
- mixed, containing all other data, namely
  - simple arrays of
    - functions
    - symbols
  - all nested arrays
    - box
    - function
    - symbol

Because general types are used mostly to indicate inclusion in the domains of functions and most functions accept empty arrays of any type, all empty arrays are included in each general type. (Although acceptance of empty arrays can cause anomalies like a character result for Add, such results are unlikely in fact to be created; if they do arise, they will probably be accepted by any function to which they are presented. For efficiency, the (empty) result of a mathematical function for a Null is whatever is most convenient for the function: Null, integer, or floating point.)