Array slicing

In computer programming, **array slicing** is an operation that extracts a subset of elements from an array and packages them as another array, possibly in a different dimension from the original.

Common examples of array slicing are extracting a substring from a string of characters, the "*ell*" in "h*ell*o", extracting a row or column from a two-dimensional array, or extracting a vector from a matrix.

Depending on the programming language, an array slice can be made out of non-consecutive elements. Also depending on the language, the elements of the new array may be aliased to (i.e., share memory with) those of the original array.

Contents

Details

For "one-dimensional" (single-indexed) arrays — vectors, sequence, strings etc. — the most common slicing operation is extraction of zero or more consecutive elements. Thus, if we have a vector containing elements (2, 5, 7, 3, 8, 6, 4, 1), and we want to create an array slice from the 3rd to the 6th items, we get (7, 3, 8, 6). In programming languages that use a 0-based indexing scheme, the slice would be from index *2* to *5*.

Reducing the range of any index to a single value effectively eliminates that index. This feature can be used, for example, to extract one-dimensional slices (vectors: in 3D, rows, columns, and tubes[1]) or two-dimensional slices (rectangular matrices) from a three-dimensional array. However, since the range can be specified at run-time, type-checked languages may require an explicit (compile-time) notation to actually eliminate the trivial indices.

General array slicing can be implemented (whether or not built into the language) by referencing every array through a dope vector or *descriptor* — a record that contains the address of the first array element, and then the range of each index and the corresponding coefficient in the indexing formula. This technique also allows immediate array transposition, index reversal, subsampling, etc. For languages like C, where the indices always start at zero, the dope vector of an array with d indices has at least $1 + 2d$ parameters. For languages that allow arbitrary lower bounds for indices, like Pascal, the dope vector needs $1 + 3d$ entries.

If the array abstraction does not support true negative indices (as for example the arrays of Ada and Pascal do), then negative indices for the bounds of the slice for a given dimension are sometimes used to specify an offset from the end of the array in that dimension. In 1-based schemes, -1 generally would indicate the second-to-last item, while in a 0-based system, it would mean the very last item.

History

The concept of slicing was surely known even before the invention of compilers. Slicing as a language feature probably started with FORTRAN (1957), more as a consequence of non-existent type and range checking than by design. The concept was also alluded to in the preliminary report for the IAL (ALGOL 58) in that the syntax allowed one or more indices of an array element (or, for that matter, of a procedure call) to be omitted when used as an actual parameter.

Kenneth Iverson's APL (1957) had very flexible multi-dimensional array slicing, which contributed much to the language's expressive power and popularity.

ALGOL 68 (1968) introduced comprehensive multi-dimension array slicing and trimming features.

Array slicing facilities have been incorporated in several modern languages, such as Ada 2005, Boo, Cobra, D, Fortran 90, Go, Rust, Matlab, Perl, Python, S-Lang, Windows PowerShell and the mathematical/statistical languages GNU Octave, S and R.

Timeline of slicing in various programming languages

1966: Fortran 66

The Fortran 66 programmers were only able to take advantage of slicing matrices by row, and then only when passing that row to a subroutine:

```
      SUBROUTINE PRINT V(VEC, LEN)
        REAL VEC(*)
        PRINT *, (VEC(I), I = 1, LEN)
      END
```

```
      PROGRAM MAIN
        PARAMETER(LEN = 3)
        REAL MATRIX(LEN, LEN)
        DATA MATRIX/1, 1, 1, 2, 4, 8, 3, 9, 27/
        CALL PRINT V(MATRIX(1, 2), LEN)
      END
```

Result:

```
   2.000000       4.000000       8.000000
```

Note that there is no dope vector in FORTRAN 66 hence the length of the slice must also be passed as an argument - or some other means - to the `SUBROUTINE`. 1970s Pascal and C had similar restrictions.

1968: Algol 68

Algol68 final report contains an early example of slicing, slices are specified in the form:

```
[lower bound:upper bound] ¢ for computers with extended character sets ¢
```

or:

```
(LOWER BOUND..UPPER BOUND) # FOR COMPUTERS WITH ONLY 6 BIT CHARACTERS. #
```

Both bounds are inclusive and can be omitted, in which case they default to the declared array bounds. Neither the stride facility, nor diagonal slice aliases are part of the revised report.

Examples:

```
[3, 3]real a := ((1, 1, 1), (2, 4, 8), (3, 9, 27)); # declaration of a variable matrix #
[,]  real c = ((1, 1, 1), (2, 4, 8), (3, 9, 27));   # constant matrix, the size is implied #
```

```
ref[]real row := a[2,];                    # alias/ref to a row slice #
ref[]real col2 = a[, 2];                   # permanent alias/ref to second column #
```

```
print ((a[:, 2], newline));                # second column slice #
print ((a[1[a, :], newline));              # last row slice #
print ((a[:, 2[a], newline));              # last column slice #
print ((a[:2, :2], newline));              # leading 2-by-2 submatrix "slice" #
```

```
+1.0000₁₀+0 +4.0000₁₀+0 +9.0000₁₀+0
+3.0000₁₀+0 +9.0000₁₀+0 +2.7000₁₀+1
+1.0000₁₀+0 +8.0000₁₀+0 +2.7000₁₀+1
+1.0000₁₀+0 +1.0000₁₀+0 +2.0000₁₀+0 +4.0000₁₀+0
```

1970s: MATLAB

```
> A = round(rand(3, 4, 5)*10) % 3x4x5 three-dimensional or cubic array
> A(:, :, 3) % 3x4 two-dimensional array along first and second dimensions

ans =

  8  3  5  7
  8  9  1  4
  4  4  2  5
```

```
> A(:, 2:3, 3) % 3x2 two-dimensional array along first and second dimensions

ans =

  3 5
  9 1
  4 2

> A(2:end, :, 3) % 2x4 two-dimensional array using the 'end' keyword; works with GNU Octave
3.2.4

ans =

   6    1    4    6
  10    1    3    1

> A(1, :, 3) % single-dimension array along second dimension

ans =

  8  3  5  7

> A(1, 2, 3) % single value
ans = 3
```

1976: S/R

Arrays in S and GNU R are always one-based, thus the indices of a new slice will begin with *one* for each dimension, regardless of the previous indices. Dimensions with length of *one* will be dropped (unless drop = FALSE). Dimension names (where present) will be preserved.

```
> A <- array(1:60, dim = c(3, 4, 5)) # 3x4x5 three-dimensional or cubic array
> A[, , 3] # 3x4 two-dimensional array along first and second dimensions
     [, 1] [, 2] [, 3] [, 4]
[1,]   25   28   31   34
[2,]   26   29   32   35
[3,]   27   30   33   36
> A[, 2:3, 3, drop = FALSE] # 3x2x1 cubic array subset (preserved dimensions)
, , 1

     [, 1] [, 2]
[1,]   28   31
[2,]   29   32
[3,]   30   33
> A[, 2, 3]  # single-dimension array along first dimension
[1] 28 29 30
> A[1, 2, 3] # single value
[1] 28
```

1977: Fortran 77

The Fortran 77 standard introduced the ability to slice and concatenate strings:

```
PROGRAM MAIN
  PRINT *, 'ABCDE'(2:4)
END
```

Produces:

```
BCD
```

Such strings could be passed by reference to another subroutine, the length would also be passed transparently to the subroutine as a kind of **short** dope vector.

```
SUBROUTINE PRINT S(STR)
  CHARACTER *(*)STR
  PRINT *, STR
END

PROGRAM MAIN
  CALL PRINT S('ABCDE'(2:4))
END
```

Again produces:

```
BCD
```

1979: Sinclair_BASIC ZX80/81/Spectrum

The standard ROM of the ZX80/81/Spectrum offers BASIC with the ability to slice and concatenate strings:

in the command part (x TO y) which points out the needed array slice the x and y value can be omitted giving the meaning to use all chained array cells (FROM x TO end) or (begin TO y). With multidimensional arrays, the slicing is only possible with the last level dimension.

```
10 LET a$="ABCDE"(2 to 4)
20 PRINT a$
```

Produces:

```
BCD
```

```
10 LET a$="ABCDE"
20 LET b$=a$(4 TO)+a$(2 TO 3)+a$(1)
30 PRINT b$
```

Produces:

```
DEBCA
```

1983: Ada 83 and above

Ada 83 supports slices for all array types. Like Fortran 77 such arrays could be passed by reference to another subroutine, the length would also be passed transparently to the subroutine as a kind of **short** dope vector.

```
with Text_IO;

procedure Main is
   Text : String := "ABCDE";
begin
   Text_IO.Put_Line (Text (2 .. 4));
end Main;
```

Produces:

```
BCD
```

Note: Since in Ada indices are n-based the term `Text (2 .. 4)` will result in an Array with the base index of 2.

The definition for `Text_IO.Put_Line` is:

```
package Ada.Text_IO is

   procedure Put_Line(Item : in  String);
```

The definition for `String` is:

```
package Standard is

   subtype Positive is Integer range 1 .. Integer'Last;

   type String is array(Positive range <>) of Character;
   pragma Pack(String);
```

As Ada supports true negative indices as in `type History_Data_Array is array (-6000 .. 2010) of History_Data;` it places no special meaning on negative indices. In the example above the term `Some_History_Data (-30 .. 30)` would slice the `History_Data` from 31 BC to 30 AD (since there was no year zero, the year number 0 actually refers to 1 BC).

1987: Perl

If we have

```
@a = (2, 5, 7, 3, 8, 6, 4);
```

as above, then the first 3 elements, middle 3 elements and last 3 elements would be:

```
@a[0..2];   # (2, 5, 7)
@a[2..4];   # (7, 3, 8)
@a[-3..-1]; # (8, 6, 4)
```

Perl supports negative list indices. The -1 index is the last element, -2 the penultimate element, etc. In addition, Perl supports slicing based on expressions, for example:

```
@a[ 3.. $#a ];   # 4th element until the end (3, 8, 6, 4)
@a[ grep { !($_ % 3) } (0...$#a) ];    # 1st, 4th and 7th element (2,3,4)
@a[ grep { !(($_+1) % 3) } (0..$#a) ]; # every 3rd element (7,6)
```

1991: Python

If you have the following list:

```
>>> nums = [1, 3, 5, 7, 8, 13, 20]
```

Then it is possible to slice by using a notation similar to element retrieval:

```
>>> nums[3]   # no slicing
7
>>> nums[:3]  # from index 0 (inclusive) until index 3 (exclusive)
[1, 3, 5]
>>> nums[1:5]
[3, 5, 7, 8]
>>> nums[-3:]
[8, 13, 20]
```

Note that Python allows negative list indices. The index -1 represents the last element, -2 the penultimate element, etc. Python also allows a step property by appending an extra colon and a value. For example:

```
>>> nums[3:]
[7, 8, 13, 20]
>>> nums[3::] # == nums[3:]
[7, 8, 13, 20]
>>> nums[::3] # starting at index 0 and getting every third element
[1, 7, 20]
>>> nums[1:5:2] # from index 1 until index 5 and getting every second element
[3, 7]
```

The stride syntax (`nums[1:5:2]`) was introduced in the second half of the 1990s, as a result of requests put forward by scientific users in the Python "matrix-SIG" (special interest group).[2]

Slice semantics potentially differ per object; new semantics can be introduced when operator overloading the indexing operator. With Python standard lists (which are dynamic arrays), every slice is a copy. Slices of NumPy arrays, by contrast, are views onto the same underlying buffer.

1992: Fortran 90 and above

In Fortran 90, slices are specified in the form

```
lower_bound:upper_bound[:stride]
```

Both bounds are inclusive and can be omitted, in which case they default to the declared array bounds. Stride defaults to 1. Example:

```
real, dimension(m, n):: a  ! declaration of a matrix

print *, a(:, 2) ! second column
print *, a(m, :) ! last row
print *, a(:10, :10) ! leading 10-by-10 submatrix
```

1994: Analytica

Each dimension of an array value in Analytica is identified by an Index variable. When slicing or subscripting, the syntax identifies the dimension(s) over which you are slicing or subscripting by naming the dimension. Such as:

```
Index I := 1..5   { Definition of a numerical Index }
Index J := ['A', 'B', 'C'] { Definition of a text-valued Index }
Variable X := Array(I, J, [[10, 20, 30], [1, 2, 3], ....]) { Definition of a 2D value }
X[I = 1, J = 'B']  -> 20  { Subscript to obtain a single value }
X[I = 1] ->  Array(J, [10, 20, 30])  { Slice out a 1D array. }
X[J = 2] -> Array(I, [20, 2, ....]) { Slice out a 1D array over the other dimension. }
X[I = 1..3]  {Slice out first four elements over I with all elements over J}
```

Naming indexes in slicing and subscripting is similar to naming parameters in function calls instead of relying on a fixed sequence of parameters. One advantage of naming indexes in slicing is that the programmer does not have to remember the sequence of Indexes, in a multidimensional array. A deeper advantage is that expressions generalize automatically and safely without requiring a rewrite when the number of dimensions of X changes.

1998: S-Lang

Array slicing was introduced in version 1.0. Earlier versions did not support this feature.

Suppose that A is a 1-d array such as

```
    A = [1:50];           % A = [1, 2, 3, ...49, 50]
```

Then an array B of first 5 elements of A may be created using

```
    B = A[[:4]];
```

Similarly, B may be assigned to an array of the last 5 elements of A via:

```
    B = A[[-5:]];
```

Other examples of 1-d slicing include:

```
    A[-1]                 % The last element of A
    A[*]                  % All elements of A
    A[[::2]]              % All even elements of A
    A[[1::2]]             % All odd elements of A
    A[[-1::-2]]           % All even elements in the reversed order
    A[[[0:3], [10:14]]]   % Elements 0-3 and 10-14
```

Slicing of higher-dimensional arrays works similarly:

```
    A[-1, *]              % The last row of A
    A[[1:5], [2:7]]       % 2d array using rows 1-5 and columns 2-7
    A[[5:1:-1], [2:7]]    % Same as above except the rows are reversed
```

Array indices can also be arrays of integers. For example, suppose that `I = [0:9]` is an array of 10 integers. Then `A[I]` is equivalent to an array of the first 10 elements of `A`. A practical example of this is a sorting operation such as:

```
    I = array_sort(A);    % Obtain a list of sort indices
    B = A[I];             % B is the sorted version of A
    C = A[array_sort(A)]; % Same as above but more concise.
```

1999: D

Consider the array:

```
int[] a = [2, 5, 7, 3, 8, 6, 4, 1];
```

Take a slice out of it:

```
int[] b = a[2 .. 5];
```

and the contents of `b` will be `[7, 3, 8]`. The first index of the slice is inclusive, the second is exclusive.

```
auto c = a[$ - 4 .. $ - 2];
```

means that the dynamic array `c` now contains `[8, 6]` because inside the [] the `$` symbol refers to the length of the array.

D array slices are aliased to the original array, so:

```
b[2] = 10;
```

means that `a` now has the contents `[2, 5, 7, 3, 10, 6, 4, 1]`. To create a copy of the array data, instead of only an alias, do:

```
auto b = a[2 .. 5].dup;
```

Unlike Python, D slice bounds don't saturate, so code equivalent to this Python code is an error in D:

```
>>> d = [10, 20, 30]
>>> d[1 : 5]
[20, 30]
```

2004: SuperCollider

The programming language SuperCollider implements some concepts from J/APL. Slicing looks as follows:

```
a = [3, 1, 5, 7]            // assign an array to the variable a
a[0..1]                     // return the first two elements of a
a[..1]                      // return the first two elements of a: the zero can be omitted
a[2..]                      // return the element 3 till last one
a[[0, 3]]                   // return the first and the fourth element of a

a[[0, 3]] = [100, 200]      // replace the first and the fourth element of a
a[2..] = [100, 200]         // replace the two last elements of a

// assign a multidimensional array to the variable a
a = [[0, 1, 2, 3, 4], [5, 6, 7, 8, 9], [10, 11, 12, 13, 14], [15, 16, 17, 18, 19]];
a.slice(2, 3);              // take a slice with coordinates 2 and 3 (returns 13)
a.slice(nil, 3);            // take an orthogonal slice (returns [3, 8, 13, 18])
```

2005: fish

Arrays in fish are always one-based, thus the indices of a new slice will begin with *one*, regardless of the previous indices.

```
> set A (seq 3 2 11)       # $A is an array with the values 3, 5, 7, 9, 11

> echo $A[(seq 2)]         # Print the first two elements of $A
3 5

> set B $A[1 2]            # $B contains the first and second element of $A, i.e. 3, 5

> set -e A[$B]; echo $A    # Erase the third and fifth elements of $A, print $A
3 5 9
```

2006: Cobra

Cobra supports Python-style slicing. If you have a list

```
nums = [1, 3, 5, 7, 8, 13, 20]
```

, then the first 3 elements, middle 3 elements, and last 3 elements would be:

```
nums[:3]   # equals [1, 3, 5]
nums[2:5]  # equals [5, 7, 8]
nums[-3:]  # equals [8, 13, 20]
```

Cobra also supports slicing-style syntax for 'numeric for loops':

```
for i in 2 : 5
    print i
# prints 2, 3, 4

for j in 3
    print j
# prints 0, 1, 2
```

2006: Windows PowerShell

Arrays are zero-based in PowerShell and can be defined using the comma operator:

```
PS  > $a = 2, 5, 7, 3, 8, 6, 4, 1
PS  > # Print the first two elements of $a:
PS  > "$($a[0, 1])"
2 5
PS  > # Take a slice out of it using the range operator:
PS  > "$($a[2..5])"
7 3 8 6
PS  > # Get the last 3 elements:
PS  > "$($a[-3..-1])"
6 4 1
PS  > # Return the content of the array in reverse order:
PS  > "$($a[($a.Length - 1)..0])" # Length is a property of System.Object[]
1 4 6 8 3 7 5 2
```

2009: Go

Go supports Python-style syntax for slicing (except negative indices are not supported). Arrays and slices can be sliced. If you have a slice

```
nums := []int{1, 3, 5, 7, 8, 13, 20}
```

then the first 3 elements, middle 3 elements, last 3 elements, and a copy of the entire slice would be:

```
nums[:3]  // equals []int{1, 3, 5}
nums[2:5] // equals []int{5, 7, 8}
nums[4:]  // equals []int{8, 13, 20}
nums[:]   // equals []int{1, 3, 5, 7, 8, 13, 20}
```

Slices in Go are reference types, which means that different slices may refer to the same underlying array.

2010: Cilk Plus

Cilk Plus supports syntax for array slicing as an extension to C and C++.

```
array_base [lower_bound:length[:stride]]*
```

Cilk Plus slicing looks as follows:

```
A[:]     // All of vector A
B[2:6]   // Elements 2 to 7 of vector B
C[:][5]  // Column 5 of matrix C
D[0:3:2] // Elements 0, 2, 4 of vector D
```

Cilk Plus's array slicing differs from Fortran's in two ways:

- the second parameter is the length (number of elements in the slice) instead of the upper bound, in order to be consistent with standard C libraries;
- slicing never produces a temporary, and thus never needs to allocate memory. Assignments are required to be either non-overlapping or perfectly overlapping, otherwise the result is undefined.

2012: Julia

Julia array slicing (https://web.archive.org/web/20160123232725/http://docs.julialang.org/en/release-0.4/manual/arrays/) is like that of Matlab, but uses square brackets. Example:

```
julia> x = rand(4, 3)
4x3 Array{Float64,2}:
 0.323877  0.186253  0.600605
 0.404664  0.894781  0.0955007
 0.223562  0.18859   0.120011
 0.149316  0.779823  0.0690126

julia> x[:, 2]                # get the second column.
4-element Array{Float64,1}:
 0.186253
 0.894781
```

```
 0.18859
 0.779823

julia> x[1, :]                # get the first row.
1x3 Array{Float64,2}:
 0.323877  0.186253  0.600605

julia> x[1:2,2:3]             # get the submatrix spanning rows 1,2 and columns 2,3
2x2 Array{Float64,2}:
 0.186253  0.600605
 0.894781  0.0955007
```

See also

- Comparison of programming languages (array)#Slicing

References

1. Zhang, Zemin; Aeron, Shuchin (2017-03-15). "Exact Tensor Completion Using t-SVD" (https://doi.org/10.1109%2Ftsp.2016.2639466). *IEEE Transactions on Signal Processing*. Institute of Electrical and Electronics Engineers (IEEE). **65** (6): 1511–1526. doi:10.1109/tsp.2016.2639466 (https://doi.org/10.1109%2Ftsp.2016.2639466). ISSN 1053-587X (https://www.worldcat.org/issn/1053-587X).
2. Millman, K. Jarrod; Aivazis, Michael (2011). "Python for Scientists and Engineers" (http://www.computer.org/csdl/mags/cs/2011/02/mcs2011020009.html). *Computing in Science and Engineering*. **13** (2): 9–12.