2e. Arrays (Vectors and Matrices)

Martin Alfaro
PhD in Economics

INTRODUCTION

So far, we've explored scalar variables, which store single-element objects. Now, we'll shift our focus to **collections**, which correspond to **variables comprising multiple elements**. Julia provides several forms of collections, including:

- Arrays (including vectors and matrices)
- Tuples and Named Tuples
- Dictionaries
- Sets

Arrays represent one of the most common data structures for collections. Their distinctive feature is that all their elements share a **homogeneous type**. Formally, arrays are objects with type `Array{T,d}`, where `d` is the number of dimensions and `T` is the element type (e.g., `Int64` or `Float64`). Two special categories of arrays are:

- **Vectors**: 1-dimensional arrays. They're represented by the type `Vector{T}`, which is an alias for `Array{T,1}`.
- **Matrices**: 2-dimensional arrays. They're represented by the type `Matrix{T}`, which is an alias for `Array{T,2}`.

Although we provide a subsection about matrices at the end, this is labeled as optional. The reason is that vectors are sufficient for conveying the topics of this website.

> **Remark**
>
> **Julia uses 1 as an array's first index**. This contrasts with many other languages (e.g., Python), where 0 is the first index.

VECTORS

Vectors in Julia are defined as *column-vectors*, and their elements are separated by a comma or a semicolon.

```
x = [1, 2, 3]          #= column-vector (defined using commas or semicolons)
                           Vector{Int64} (alias for Array{Int64, 1}) =#

x = [1; 2; 3]          # equivalent notation to define `x`
```

```
julia> x
3-element Vector{Int64}:
 1
 2
 3
```

Note that, as previously indicated, arrays are defined so that all elements share a common type. This requirement, however, doesn't prevent arrays from holding elements of different kinds, such as numbers and strings. The reason is that the shared type can be an abstract type, effectively providing a unifying type for the collection.

For example, defining the vector `[1, 2.5, "Hello"]` is valid, because its elements will be assigned the abstract type `Any`. Recall that `Any` sits at the top of Julia's type hierarchy, encompassing all concrete types supported by the language.

Although arrays that mix element types are possible, they're strongly discouraged for several reasons, including performance.

ACCESSING VECTOR ELEMENTS

Given a vector `x`, we can access its *i*-th element with `x[i]` and retrieve all its elements with `x[:]`.

```
x = [4, 5, 6]
```

```
julia> x
3-element Vector{Int64}:
 4
 5
 6

julia> x[2]
5

julia> x[:]
3-element Vector{Int64}:
 4
 5
 6
```

It's also possible to access a subset of elements within `x`. There are several approaches to achieve this, and we'll only present two basic ones at this point. The simplest method involves setting the indices **via a vector**, using the syntax `x[<vector>]`.

```
x = [4, 5, 6, 7, 8]
```

```
julia> x
5-element Vector{Int64}:
 4
 5
 6
 7
 8

julia> x[[1,3]] # elements of 'x' with indices 1 and 3
2-element Vector{Int64}:
 4
 6

julia> x[1,3] # be careful! this is the notation used for matrices, indicating 'x[row 1, column 3]'
ERROR: BoundsError: attempt to access 5-element Vector{Int64} at index [1, 3]
```

The second approach sets the indices **via ranges**. These are denoted as `<first>:<steps>:<last>`, with Julia assuming unit increments when omitting `<steps>`. To express the first and last index in a range, Julia provides the keywords `begin` and `end`.

```
x = [4, 5, 6, 7, 8]
```

```
julia> x
5-element Vector{Int64}:
 4
 5
 6
 7
 8


julia> x[1:2] # steps with unit increments (default increments)
2-element Vector{Int64}:
 4
 5

julia> x[1:2:5] # steps with increments of 2 (explicit increments required)
3-element Vector{Int64}:
 4
 6
 8

julia> x[begin:end] # all elements equivalent to 'x[:]' or 'x[1:end]'
5-element Vector{Int64}:
 4
 5
 6
 7
 8
```

MATRICES (*OPTIONAL*)

Matrices can be defined as collections of row or column vectors. When constructing a matrix from multiple row vectors, each row must be separated by a semicolon `;`. Conversely, if created using multiple column vectors, their elements are separated by spaces.

Note that row vectors are treated as special cases of matrices, with their elements separated by a space. In this form, they're matrices consisting of a single row and multiple columns.

```
X = [1 2 ; 3 4]       #= matrix as a collection of row-vectors, separated by semicolons
                         Matrix{Int64} (alias for Array{Int64, 2})=#

X = [ [1,3] [2,4] ]   # identical to `X`, but defined through a collection of column-vectors

Y = [1 2 3]           #= row-vector (defined without commas)
                         Matrix{Int64} (alias for Array{Int64, 2}) =#
```

```
julia> X
2×2 Matrix{Int64}:
 1  2
 3  4

julia> Y
1×3 Matrix{Int64}:
 1  2  3
```

ACCESSING MATRIX ELEMENTS

Given a matrix `X`, we can access the element at row `r` and column `c` by `X[r,c]`. Moreover, `X[r,:]` selects all elements across the row `r`, while `X[:,c]` selects all elements of column `c`. For a row vector `Y`, its *i*-th element can be accessed with `X[i]`. [1]

```
X = [5 6 ; 7 8] # matrix

Y = [4 5 6]     # row-vector
```

```
julia> X
2×2 Matrix{Int64}:
 5  6
 7  8


julia> X[2,1]
7


julia> X[1,:]
2-element Vector{Int64}:
 5
 6
julia> X[:,2]
2-element Vector{Int64}:
 6
 8
julia> Y[2]
5
```

To access a subset of elements within a matrix, use the same methods employed for vectors. Their only difference is that these methods must be applied to rows or columns.

```
X = [5 6 ; 7 8]
```

```
julia> X
2×2 Matrix{Int64}:
 5  6
 7  8


julia> X[[1,2],1]
2-element Vector{Int64}:
 5
 7


julia> X[1:2,1]
2-element Vector{Int64}:
 5
 7
julia> X[begin:end,1]
2-element Vector{Int64}:
 5
 7
```

FOOTNOTES

[1] Matrices also support linear indexing. For example, a 3×3 matrix `X` accepts indices from 1 to 9. However, unless you intend to iterate over all elements, the two-dimensional notation `X[r,c]` is generally clearer and easier to interpret.