# *2e.* Arrays (Vectors and Matrices)

Martin Alfaro
PhD in Economics

## INTRODUCTION

So far, we've explored variables representing single-element objects. Now, we'll shift our focus to **collections**, defined as **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. They are formally defined as objects with type `Array{T,d}`, where `d` is the array's dimension and `T` is its elements' type (e.g., `Int64` or `Float64`).

Two special categories of arrays are **vectors** (1-dimensional arrays) and **matrices** (2-dimensional arrays). Vectors are represented by the type `Vector{T}`, which is an alias for `Array{T,1}`. For its part, matrices use 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.

```julia
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
```

> **Remark**
>
> Arrays can hold elements of various types, such as numbers and strings. For example, `[1, 2.5, "Hello"]` is a valid vector in Julia, identifying its elements as having type `Any` (recall that `Any` encompasses all the possible types supported by Julia). While arrays mixing types can be created, they're highly 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[:]`.

```julia
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 `x`'s elements. 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 increments of one if we omit `<steps>`. To respectively express the first and last index in a range, you can use 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. If they're created through multiple row vectors, each row has to be separated by a semicolon `;`. If we instead adopt multiple column vectors, their elements need to be separated by a space.

Note that row vectors are considered as special cases of matrices, with their elements separated by a space—they're matrices with multiple columns having one element.

```
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 its element at row `r` and column `c` by `X[r,c]`. Moreover, we can select all elements across the row `r` by `X[r,:]`, and all elements of column `c` by `X[:,c]`. When, instead, we have a row vector `Y`, its i-th element can be accessed using `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, you must follow the same approaches as with vectors, but applied to either 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] We could also use this approach for matrices, as they accept a linear index. For instance, a 3x3 matrix `X` accepts indices between 1 and 9. However, unless you want to iterate over all elements of a matrix, the notation `X[r,c]` is easier to interpret.