Vectors, Arrays and Matrices

HA Van Thao
Faculty of Math & Computer Science, HCMUS

Contents

Array Basics
Shape and Dimension

```
julia> a = [10, 20, 30]
3-element Array{Int64,1}:
 10
 20
 30
```

```
julia> a = ["foo", "bar", 10]
3-element Array{Any,1}:
   "foo"
   "bar"
 10
```

- The arrays are of types `Array{Int64,1}` and `Array{Any,1}` respectively
- The `1` in `Array{Int64,1}` and `Array{Any,1}` indicates that the array is one dimensional

Array Basics
Shape and Dimension

```
julia> typeof(randn(100))
Array{Float64,1}
```

- To say that an array is one dimensional is to say that it is flat
- We can also confirm that a is flat using the `size()` or `ndims()` functions

```
julia> size(a)
(3,)

julia> ndims(a)
1
```


Ha Van Thao (hvthao@gmail.com)

Array Basics
Shape and Dimension

- To create a two-dimensional array

```
julia> eye(3)
3x3 Array{Float64,2}:
 1.0  0.0  0.0
 0.0  1.0  0.0
 0.0  0.0  1.0

julia> diagm([2, 4])
2x2 Array{Int64,2}:
 2  0
 0  4

julia> size(eye(3))
(3,3)
```


Ha Van Thao (hvthao@gmail.com)

Array Basics
Array vs Vector vs Matrix

- In Julia, in addition to arrays you will see the types `Vector` and `Matrix`
- However, these are just aliases for one- and two-dimensional arrays respectively

```
julia> Array{Int64, 1} == Vector{Int64}
true

julia> Array{Int64, 2} == Matrix{Int64}
true

julia> Array{Int64, 1} == Matrix{Int64}
false

julia> Array{Int64, 3} == Matrix{Int64}
false
```


Ha Van Thao (hvthao@gmail.com)

Array Basics
Changing Dimensions

- The primary function for changing the dimension of an array is `reshape()`

```
julia> a = [10, 20, 30, 40]
4-element Array{Int64,1}:
 10
 20
 30
 40

julia> b = reshape(a, 2, 2)
2x2 Array{Int64,2}:
 10  30
 20  40

julia> b
2x2 Array{Int64,2}:
 10  30
 20  40
```

Array Basics
Changing Dimensions

- Notice that `reshape()` returns a "view" on the existing array
- This means that changing the data in the new array will modify the data in the old one

```
julia> b[1, 1] = 100  # Continuing the previous example
100

julia> b
2x2 Array{Int64,2}:
 100  30
  20  40

julia> a  # First element has changed
4-element Array{Int64,1}:
 100
  20
  30
  40
```

Array Basics
Changing Dimensions

- To collapse an array along one dimension you can use `squeeze()`

```
julia> a = [1 2 3 4]   # Two dimensional
1x4 Array{Int64,2}:
 1  2  3  4

julia> squeeze(a, 1)
4-element Array{Int64,1}:
 1
 2
 3
 4
```

- The return value is an Array with the specified dimension "flattened"

Array Basics
Why Flat Arrays?

- As we've seen, in Julia we have both
  - one-dimensional arrays (i.e., flat arrays)
  - arrays of size (1, n) or (n, 1) that represent row and column vectors respectively
- Why do we need both?
  - On one hand, dimension matters when we come to matrix algebra
  - On the other, we use arrays in many settings that don't involve matrix algebra
  - In such cases, we don't care about the distinction between row and column vectors
  - This is why many Julia functions return flat arrays by default

Creating Arrays
Functions that Return Arrays

- We've already seen some functions for creating arrays

```
julia> eye(2)
2x2 Array{Float64,2}:
 1.0  0.0
 0.0  1.0

julia> zeros(3)
3-element Array{Float64,1}:
 0.0
 0.0
 0.0
```

- You can create an empty array using the `Array()` constructor

```
julia> x = Array(Float64, 2, 2)
2x2 Array{Float64,2}:
 0.0            2.82622e-316
 2.76235e-318   2.82622e-316
```


Ha Van Thao (hvthao@gmail.com)

Creating Arrays
Functions that Return Arrays

- Other important functions that return arrays are

```
julia> ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0


julia> fill("foo", 2, 2)
2x2 Array{ASCIIString,2}:
 "foo"  "foo"
 "foo"  "foo"
```

Creating Arrays
Manual Array Definitions

- You can create one dimensional arrays from specified data

```
julia> a = [10, 20, 30, 40]
4-element Array{Int64,1}:
 10
 20
 30
 40
```

- In two dimensions we can proceed as follows

```
julia> a = [10 20 30 40]  # Two dimensional, shape is 1 x n
1x4 Array{Int64,2}:
 10  20  30  40

julia> ndims(a)
2

julia> a = [10 20; 30 40]  # 2 x 2
2x2 Array{Int64,2}:
 10  20
 30  40
```

Creating Arrays
Manual Array Definitions

- You might then assume that `a = [10; 20; 30; 40]` creates a two dimensional column vector but unfortunately this isn't the case

```
julia> a = [10; 20; 30; 40]
4-element Array{Int64,1}:
 10
 20
 30
 40

julia> ndims(a)
1
```

- Instead transpose the row vector

```
julia> a = [10 20 30 40]'
4x1 Array{Int64,2}:
 10
 20
 30
 40
```

Creating Arrays
Array Indexing

- We've already seen the basics of array indexing

```
julia> a = collect(10:10:40)
4-element Array{Int64,1}:
 10
 20
 30
 40

julia> a[end-1]
30

julia> a[1:3]
3-element Array{Int64,1}:
 10
 20
 30
```

Creating Arrays
Array Indexing

- For 2D arrays the index syntax is straightforward

```
julia> a = randn(2, 2)
2x2 Array{Float64,2}:
 1.37556  0.924224
 1.52899  0.815694

julia> a[1, 1]
1.37555922478634

julia> a[1, :]  # First row
1x2 Array{Float64,2}:
 1.37556  0.924224

julia> a[:, 1]  # First column
2-element Array{Float64,1}:
 1.37556
 1.52899
```

Creating Arrays
Array Indexing

- Booleans can be used to extract elements

```
julia> a = randn(2, 2)
2x2 Array{Float64,2}:
 -0.121311  0.654559
 -0.297859  0.89208

julia> b = [true false; false true]
2x2 Array{Bool,2}:
  true  false
 false   true

julia> a[b]
2-element Array{Float64,1}:
 -0.121311
  0.89208
```


Ha Van Thao (hvthao@gmail.com)

Creating Arrays
Array Indexing

- Some or all elements of an array can be set equal to one number using slice notation

```
julia> a = Array(Float64, 4)
4-element Array{Float64,1}:
 1.30822e-282
 1.2732e-313
 4.48229e-316
 1.30824e-282

julia> a[2:end] = 42
42

julia> a
4-element Array{Float64,1}:
  1.30822e-282
 42.0
 42.0
 42.0
```

Creating Arrays
Passing Arrays

```
julia> a = ones(3)
3-element Array{Float64,1}:
 1.0
 1.0
 1.0

julia> b = a
3-element Array{Float64,1}:
 1.0
 1.0
 1.0

julia> b[3] = 44
44

julia> a
3-element Array{Float64,1}:
  1.0
  1.0
 44.0
```

- As in Python, all arrays are passed by reference
- What this means is that if a is an array and we set b = a then a and b point to exactly the same data
- Hence any change in b is reflected in a

Creating Arrays
Passing Arrays

```
julia> a = ones(3)
3-element Array{Float64,1}:
 1.0
 1.0
 1.0

julia> b = copy(a)
3-element Array{Float64,1}:
 1.0
 1.0
 1.0

julia> b[3] = 44
44

julia> a
3-element Array{Float64,1}:
 1.0
 1.0
 1.0
```

- It's very inefficient to copy arrays unnecessarily
- If you do need an actual copy in Julia, just use `copy()`

Operations on Arrays
Array Methods

- Julia provides standard functions for acting on arrays, some of which we've already seen

```
julia> a = [-1, 0, 1]
3-element Array{Int64,1}:
 -1
  0
  1

julia> length(a)
3

julia> sum(a)
0
julia> mean(a)
0.0

julia> std(a)
1.0
```

```julia
julia> var(a)
1.0

julia> maximum(a)
1

julia> minimum(a)
-1

julia> b = sort(a, rev=true)  # Returns new array, original not modified
3-element Array{Int64,1}:
  1
  0
 -1

julia> b === a  # === tests if arrays are identical (i.e share same memory)
false

julia> b = sort!(a, rev=true)  # Returns *modified original* array
3-element Array{Int64,1}:
  1
  0
 -1

julia> b === a
true
```

Operations on Arrays
Matrix Algebra

- For two dimensional arrays, * means matrix multiplication

```
julia> a = ones(1, 2)
1x2 Array{Float64,2}:
 1.0  1.0

julia> b = ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> a * b
1x2 Array{Float64,2}:
 2.0  2.0

julia> b * a'
2x1 Array{Float64,2}:
 2.0
 2.0
```

Operations on Arrays
Matrix Algebra

- To solve the linear system A X = B for X use A \ B
- The first one is numerically more stable and should be preferred in most cases

```
julia> A = [1 2; 2 3]
2x2 Array{Int64,2}:
 1  2
 2  3

julia> B = ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> A \ B
2x2 Array{Float64,2}:
 -1.0  -1.0
  1.0   1.0

julia> inv(A) * B
2x2 Array{Float64,2}:
 -1.0  -1.0
  1.0   1.0
```


Ha Van Thao (hvthao@gmail.com)

Operations on Arrays
Matrix Algebra

- If you want an inner product in this setting use `dot()`

```
julia> dot(ones(2), ones(2))
2.0
```

- Matrix multiplication using one dimensional vectors is a bit inconsistent — pre-multiplication by the matrix is OK, but post-multiplication gives an error

```
julia> b = ones(2, 2)
2x2 Array{Float64,2}:
 1.0  1.0
 1.0  1.0

julia> b * ones(2)
2-element Array{Float64,1}:
 2.0
 2.0
```

Preferences

- http://julialang.org

Ha Van Thao (hvthao@gmail.com)