A Short and Incomplete Introduction to Julia

Part 3: Vectors and Loops

Riccardo Murri <riccardo.murri@uzh.ch>
S3IT: Services and Support for Science IT,
University of Zurich

September 12, 2019

Array basics

Constructing Vectors

Vectors (1D arrays) are created and initialized by enclosing values in square brackets ‘[’ and ‘]’:

```
julia>   v = [1, 2, 3, 4, 5]
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

An empty array is created by a pair of square brackets with nothing in between:

```
julia>   z = []
0-element Array{Any,1}
```

The Type of an Array

Every value in Julia has a type
— arrays are no exception:

```
julia>   v = [1, 3, 3, 7];
julia>   typeof(v)
Array{Int64,1}
```

`Array{...}` indicates a *parametric* type:

- `Int64` is the type of elements in the array (also available as `eltype(a)`);
- `1` is the number of dimensions.

The Type of an Array

Every value in Julia has a type
— arrays are no exception:

```
julia>   v = [1, 3, 3, 7];
julia>   typeof(v)
Array{Int64,1}
```

`Array{...}` indicates a *parametric* type:

- `Int64` is the type of elements in the array (also available as `eltype(a)`);
- `1` is the number of dimensions.

The Type of an Array

Every value in Julia has a type
— arrays are no exception:

```
julia>   v = [1, 3, 3, 7];
julia>   typeof(v)
Array{Int64, 1 }
```

`Array{...}` indicates a *parametric* type:

- `Int64` is the type of elements in the array (also available as `eltype(a)`);
- `1` is the number of dimensions.

Vectors, Matrices, Arrays

Julia provides syntax for entering literal 1D and 2D arrays; note the usage of ‘ , ’ vs. ‘   ’ vs. ‘ ; ’

1D arrays are entered with commas and represented as (column) vectors:

```
julia>   v = [1 ,  2 ,  3]
3-element ArrayInt64,1:
 1
 2
 3
```

Use space instead of comma for a “wide” matrix, which is represented as a row vector:

```
julia>   w = [1  2  3]
1×3 ArrayInt64,2:
 1  2  3
```

2D array (matrix) entry requires that rows are separated by semicolons:

```
julia>   m = [1 2 ;  3 4]
2×2 ArrayInt64,2:
 1  2
 3  4
```

No literal entry syntax is there for higher-dimensional arrays.

The `reshape()` function can be used to turn a vector into an *N*-dimensional array.

Constructing arrays, I

Julia provides a few functions to provide commonly-used arrays.

`fill(x, size)`
A newly-created array filled with value `x`.

```
julia>   fill(2, (2,3))
2×3 Array{Int64,2}:
 2  2  2
 2  2  2
```

The element type of the array is determined by the type of `x`.

`zeros(type, size)`, **`ones(type, size)`**
A newly-created array filled with `0`/`1`'s.

```
julia>   ones(2)
2-element Array{Float64,1}:
 1.0
 1.0
```

`trues(size)`, `falses(size)`
Same, but with boolean `true`/`false` entries.

Argument `type` can be omitted and defaults to `Float64`.

Constructing arrays, II

`rand(type, size)`, `randn(type, size)`
A newly-created array filled with random numbers.

`rand` picks entries using a uniform distribution, `randn` uses a normal one with $\mu = 0$ and $\sigma = 1$.

`similar(a)`
Create an uninitialized array with the same element type and size as the given array `a`.

`rand(S, size)`
A newly-created array filled with entries picked uniformly at random from collection `S`.

```
julia>
 rand("julia", (2,3))
2×3 Array{Char,2}:
 'j'  'a'  'i'
 'l'  'u'  'a'
```

Accessing elements in vectors

You can access individual items in a vector using the postfix `[]` operator.

Element indices start at 1.

```
julia>   v[1]
1
julia>   v[2] * v[3]
6
```

The keyword `end` can be used to get the last item of a vector:

```
julia>   v[end]
5
```

Slices

The notation `[`k`:`l`]` is used for accessing a *slice* of an array (the items at positions k, $k+1$, ..., l).

```
julia>   v[2:4]
3-element Array{Int64,1}:
 2
 3
 4
```

Again, one can use `end` in place of the number l to mean the length of the array.

A single colon ‘`:`’ abbreviates ‘`1:end`’.

Accessing elements in arrays

For matrices and higher-dimensional arrays, indices must be separated by a comma:

```
julia>   m[1,1]
1
julia>   m[end, end]  # last item in each dim
4

julia>   a = reshape(1:27, (3,3,3)); # 3-D array
julia>   a[1,2,3]
22
```

Slicing is possible in each dimension separately; note that dimensions with singleton index are dropped:

```
julia>   a[:, 1:2, 1]  # 3x2(x1) slice
3×2 Array{Int64,2}:
 1  4
 2  5
 3  6
```

Dimensions of an array

`ndims(a)`
Return number of dimensions of an array.

```
julia>   ndims(v)
1
julia>   ndims(a)
3
```

`length(a)`
Return the number of elements in an array:

```
julia>   length(v)
5
julia>   length(a)
27
```

`size(a)`
Return tuple of maximum index in each dimension.

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

Since indexing starts at 1, valid indices into `i`-th array dimension range from 1 up to `size(a,i)` *(inclusive!)*.

Item mutation

You can replace items in an array by assigning them a new value:

```
julia>   v[2] = 9;

julia>   v
5-element Array{Int64,1}:
 1
 9
 3
 4
 5
```

Slice mutation, I

You can also replace an entire slice of an array:

```
julia>   v[2:4] = [5,6,7];
julia>   v
5-element Array{Int64,1}:
  1
  5
  6
  7
  5
```

The new slice *must* have the same length of the one it is replacing:

```
julia>   v[2:4] = [5,6,7,8,9];
ERROR: DimensionMismatch("tried to assign 5 elements
to 3 destinations")
```

Loops

`for`-loops

With the `for` statement, you can **loop over the items of a range or collection**:

```
for i in 1:3
  # loop block
  println(i*i)
end
```

To break out of a `for` loop, use the `break` statement.

To jump to the next iteration of a `for` loop, use the `continue` statement.

The `for` statement can be used to loop over elements in *any collection*.

Loop over vectors:

```
julia>
  for n in [1,2,3]
    println(n)
  end
1
2
3
```

Loop over strings:

```
julia>
  for ch in "abc"
    println(ch)
  end
a
b
c
```

Loop over matrices and arrays:

```
julia>
  for n in [1 2; 3 4]
    println(n)
  end
1
3
2
4
```

Loop over tuples:

```
julia>   for x in (4, 2)
         println(x)
       end
4
2
```

Exercise 3.A: Write a function `avg(v)` that takes a vector `v` of numbers and returns their mean value.

Exercise 3.B: Write a function `tr(m)` which, given a (square) matrix `m`, returns its *trace*, i.e., sum of the elements along the main diagonal.

Exercise 3.C: Write a function `eye(`n`)` which, given an integer $n > 0$, returns the $n \times n$ identity matrix.

Exercise 3.D: Write a function `flip(m)` which, given a matrix `m`, returns its transpose.

Vector operations

Vector operations

By virtue of having a 1D index set, Vectors support a few operations that generic array cannot.

They are quickly recalled here.

Slice mutation, II

Function `splice!` can be used to replace a slice of a *vector* with one of a different length:

```
julia>   splice!(v, 2:4, [5,6,7,8,9]);
julia>   v
7-element Array{Int64,1}:
  1
  5
  6
  7
  8
  9
  5
```

The meaning of the bang!

By convention, a “bang” character ‘`!`’ marks functions that *mutate* their first argument *in-place*.

In other words, the bang signals that something is going to be changed destructively without an easy way to undo the effects.

This convention is followed consistently in Julia’s standard library.

Operating on vectors, I

`push!(v,e), pushfirst!(v,e)`
Extend vector `v` and insert item `e` as last (`push!`) or first (`pushfirst!`).

`pop!(v), popfirst!(v)`
Remove the last (`pop!`) or first (`popfirst!`) item in vector `v` and return it.

`append!(v1,v2), prepend!(v1,v2)`
Extend `v1` by adding elements of `v2` to its start or end.

`insert!(v, index, item)`
Insert an item into `v` at the given index.

`deleteat!(v, index)`
Remove the item at the given index and return the modified vector `v`. Subsequent items are shifted to fill the resulting gap.

Reference:

- https://docs.julialang.org/en/v1/base/arrays/

Operating on vectors, II

`sort(v), sort!(v)`
Return a sorted copy (`sort`) of vector `v`, or sort it in-place `(sort!)`. Use `sort(v, rev=true)` to reverse the ordering.

`reverse(v), reverse!(v)`
Return a copy of `v` reversed from start to end; or reverse it in-place.

Reference:

- https://docs.julialang.org/en/v1/base/sort/

Exercise 3.E: Write a function `median(v)` that takes a vector `v` of numbers and returns the median value.

Exercise 3.F: Write a function `evens(v)` that takes a vector `v` of integers and returns a vector of all the even ones.

Exercise 3.G: Write a function `deviation(v, u)` that takes a vector `v` of numbers and a single value `m` returns a vector with the difference of `u` and each element `x` of `v`.

map, reduce, filter (1)

Constructing a new collection by looping over a given one and applying a function on all elements is so common that there are specialized functions for that:

`map(fn, a)`
Return a new collection formed by applying function `fn(x)` to every element `x` of array `a`.

`filter(fn, a)`
Return a new collection formed by elements `x` of array `a` for which `fn(x)` evaluates to a `true` value.

See also: https://en.wikibooks.org/wiki/Introducing_Julia/Functions#Map

map, reduce, filter (2)

`reduce(fn2, a)`
Apply *associative* 2-ary function `fn2(x,y)` to the first two items `x` and `y` of collection `a`, then apply `fn2` to the result and the third element of `L`, and so on until all elements have been processed — return the final result.

See also: https://en.wikibooks.org/wiki/Introducing_Julia/Functions#Reduce_and_folding

This is how you could rewrite Exercises 3.F and 3.G using `map` and `filter`.

```
# *** Exercise 3.F ***
function evens(v)
  filter(iseven, v)
end
```

```
# *** Exercise 3.G ***
function deviation(v, u)
  # note:  can define
  # func's in func's!
  delta(x) = abs(x-u)
  return map(delta, v)
end
```

Anonymous functions

With higher-order functions, it is sometimes convenient being able to use functions defined “on the spot”.

Julia allows an alternate syntax for functions, specifying names of parameters and an expression to compute:

```
julia>   x -> if x>0; exp(-(1/x)^2); else 0; end
#21 (generic function with 1 method)
```

Note that functions defined this way are not assigned a name! This syntax is meant to produce a function to be immediately used:

```
julia>   map((x -> iseven(x) ? "odd" : "even"),
           rand(Int64, 3))
3-element ArrayString,1:
 "odd"
 "even"
 "odd"
```

The ‘do’ notation, I

```
# turn array of int’s
# into corresponding
# Roman numerals
map(a) do num
  if num == 1
    return "I"
  elseif num == 2
    return "II"
    # ...
  end
end # close ‘do‘ block
```

Any function whose *first* argument is a function can be used with `do`: e.g., `map()`, `filter()`, etc.

The expression defined after `do` is inserted into the function call as the first argument.

The ‘do’ notation, II

```
# turn array of int’s
# into corresponding
# Roman numerals
map(a) do num
  if num == 1
    return "I"
  elseif num == 2
    return "II"
    # ...
  end
end # close ‘do‘ block
```

The **do** block must be closed by a corresponding **end**.

The ‘do’ notation, III

```
# turn array of int’s
# into corresponding
# Roman numerals
map(a) do num
  if num == 1
    return "I"
  elseif num == 2
    return "II"
    # ...
  end
end # close ‘do‘ block
```

After `do` there comes a parameter list: e.g., ‘`do x,y`’ would substitute `x` and `y` in the expression following.

The ‘do’ notation, IV

```
open("/path/to/file", "r") do file
  while !eof(file);
    line = readline(file);
    # process line
  end
end
```

‘`do`’ blocks are not just for use with `map`.

Wherever a function is the first argument, ‘`do`’ is a fit.

Type conversion

Type conversion on assignment, I

When assigning a value to an array item or slice, automatic conversion to the element type of the array is attempted:

```
julia>   v[3] = 5.0;  # converted to Int64
julia>   v
4-element Array{Int64,1}:
 1
 3
 5
 7
```

Type conversion on assignment, II

When assigning a value to an array item or slice, automatic conversion to the element type of the array is *attempted* — and it may fail!

```
julia>  v[3] = 5.5
ERROR: InexactError: Int64(5.5)

julia>  v[3] = "five"
ERROR: MethodError: Cannot `convert` an object of
type String to an object of type Int64
```

Choosing the type of array elements, I

Julia guesses the type of elements based on the initial set of items:

```
julia>   typeof([1,2,3])
Array{Int64,1}
julia>   typeof([1.0, 2.0, 3.0])
Array{Float64,1}
```

Arrays that can hold any value are possible, using the `Any` element type:

```
julia>   typeof([1, "two", 3.0])
Array{Any,1}
```

Choosing the type of array elements, II

You can override Julia's decision for element type, by specifying it before the '`[`':

```
julia>   b = Float64[1,2,3]
3-element Array{Float64,1}:
 1.0
 2.0
 3.0
```

Or you can convert afterwards:

```
julia>   c = convert(Array{Float64,1}, [1, 2, 3])
3-element Array{Float64,1}:
 1.0
 2.0
 3.0
```

Choosing the type of array elements, III

Still, individual elements must be convertible!

```
julia>   convert(Array{Float64,1}, [1, "two", 3])
ERROR: MethodError:  Cannot `convert` an object of
type String to an object of type Float64
```