# Introduction to Julia

## Lecture 12: Julia Fundamental Types

Diego de Sousa Rodrigues
diego.desousarodrigues@sciencespo.fr

SciencesPo Paris

# Julia Fundamental Types

## Overview

In Julia, **arrays** and **tuples** are the most important data type for working with numerical data. In this lecture we give more details on :

- creating and manipulating Julia arrays
- fundamental array processing operations
- basic matrix algebra
- tuples and named tuples

# Julia Fundamental Types

## Arrays

- Since it is one of the most important types, we will start with **arrays**
- Later, we will see how arrays (and all other types in Julia) are handled in a generic and extensible way

# Julia Fundamental Types

## Arrays : shape and dimension

- We've already seen some Julia arrays in action

```
a = [0, 0, 0]
```

```
b = zeros(3)
```

# Julia Fundamental Types

## Arrays : shape and dimension

- We've already seen some Julia arrays in action

  `a = [0, 0, 0]`

  `b = zeros(3)`

- What is the difference between `a` and `b`?

# Julia Fundamental Types

## Arrays : shape and dimension

- The output tells us that the arrays are of types `Array{Int64,1}` and `Array{Float64,1}` respectively
- Here `Int64` and `Float64` are types for the elements inferred by the compiler
- The 1 in `Array{Int64,1}` and `Array{Any,1}` indicates that the array is one dimensional (i.e., a Vector)
- This is the default for many Julia functions that create arrays

# Julia Fundamental Types

## Arrays : shape and dimension

- Remember : you can access the type of any object by typing

```
typeof(zeros(3))
```

# Julia Fundamental Types

## Arrays : shape and dimension

- In Julia, one dimensional vectors are best interpreted as column vectors, which we can see when we take transposes
- We can check the dimensions of `a` using `size()` and `ndims()` functions
- What do you get ?

# Julia Fundamental Types

## Arrays : shape and dimension

- In Julia, one dimensional vectors are best interpreted as column vectors, which we can see when we take transposes
- We can check the dimensions of `a` using `size()` and `ndims()` functions
- What do you get ?
- The syntax `(3,)` displays a tuple containing one element – the size along the one dimension that exists

# Julia Fundamental Types

## Array vs. Vector vs. Matrix

- In Julia, `Vector` and `Matrix` are just other names for one- and two-dimensional arrays respectively

# Julia Fundamental Types

## Array vs. Vector vs. Matrix

- In Julia, `Vector` and `Matrix` are just other names for one- and two-dimensional arrays respectively

```
Array{Int64, 1} == Vector{Int64}
Array{Int64, 2} == Matrix{Int64}
```

# Julia Fundamental Types

## Array vs. Vector vs. Matrix

- If you type `[1,2,3]` what do you get ?

# Julia Fundamental Types

## Array vs. Vector vs. Matrix

- If you type `[1,2,3]` what do you get ?
- And if you type `[1 2 3]` ? What is the difference ?

# Julia Fundamental Types

## Array vs. Vector vs. Matrix

As we've seen, in Julia we have both :

- one-dimensional arrays i.e., flat arrays or size (n, 1) (= column vector)

# Julia Fundamental Types

## Array vs. Vector vs. Matrix

As we've seen, in Julia we have both :

- one-dimensional arrays i.e., flat arrays or size (n, 1) (= column vector)
- arrays of size (1, n) that represent row vectors

Why do we need both ?

- On one hand, dimension matters for 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

# Julia Fundamental Types

## Creating Arrays

- We've already seen some functions to create a vector filled of zeros

# Julia Fundamental Types

## Creating Arrays

- We've already seen some functions to create a vector filled of zeros
- This works also with matrices and higher dimensions arrays
- What do you get when typing `zeros(2,3)` ? How many rows / columns ?

# Julia Fundamental Types

## Creating Arrays

- We've already seen some functions to create a vector filled of zeros
- This works also with matrices and higher dimensions arrays
- What do you get when typing `zeros(2,3)` ? How many rows / columns ?
- Another method is to use `fill(5.0,3,2)`. Now what do you obtain ?

# Julia Fundamental Types

## Creating Arrays

**Exercise 1 : your own function creating empty matrices**

- Create your own function `myzeros` that takes two arguments and return any vector or matrix of size (n,m). If you specify only one parameter `n`, it retuns a flat array of dimension 1 (exactly as `zeros()`) by default.
- Hint : call the function `fill` inside your function

# Julia Fundamental Types

## Creating Arrays

- Last option, use a constructor : `Array{Type}(dims)`
- `x = Array{Float64}(undef, 2, 2)`
- What do you get ? Why ?

# Julia Fundamental Types

## Creating Arrays from Existing Arrays

- For the most part, we will avoid directly specifying the types of arrays, and let the compiler deduce the optimal types on its own
- The reasons for this, discussed in more detail in this lecture, are to ensure both clarity and generality
- One place this can be inconvenient is when we need to create an array based on an existing array
- First, note that assignment in Julia binds a name to a value, but does not make a copy of that type

# Julia Fundamental Types

## Creating Arrays from Existing Arrays

- **First, note that assignment in Julia binds a name to a value, but does not make a copy of that type (!!)**

```
x = [1, 2, 3]
y = x
y[1] = 2
```

- Now what is `x` ?

# Julia Fundamental Types

## Creating Arrays from Existing Arrays

- In the above, `y = x` simply creates a new named binding called `y` which refers to whatever `x` currently binds to

# Julia Fundamental Types

## Creating Arrays from Existing Arrays

- In the above, `y = x` simply creates a new named binding called `y` which refers to whatever `x` currently binds to
- To copy the data, you need to be more explicit

```
x = [1, 2, 3]
y = copy(x)
y[1] = 2
```

- Now what is `x`?

# Julia Fundamental Types

## Creating Arrays from Existing Arrays

- However, rather than making a copy of `x`, you may want to just have a similarly sized array

```
x = [1, 2, 3]
y = similar(x)
```

# Julia Fundamental Types

## Creating Arrays from Existing Arrays

- However, rather than making a copy of `x`, you may want to just have a similarly sized array

```
x = [1, 2, 3]
y = similar(x)
```

- We can also use similar to pre-allocate a vector with a different size, but the same shape

```
x = [1, 2, 3]
y = similar(x, 10)
```

# Julia Fundamental Types

## Creating Arrays from Existing Arrays

- However, rather than making a copy of `x`, you may want to just have a similarly sized array

```
x = [1, 2, 3]
y = similar(x)
```

- We can also use similar to pre-allocate a vector with a different size, but the same shape

```
x = [1, 2, 3]
y = similar(x, 10)
```

- Or even, with different shape and size :

```
y = similar(x,2,2)
```

# Julia Fundamental Types

## Creating Arrays from Existing Arrays

- Finally, you can also create manually multidimensional arrays :

```
x = [1 2 3; 4 5 6]
```

# Julia Fundamental Types

## Array Indexing

- We've already seen the basics of array indexing. Let `x = [1, 2, 3, 4, 5]`

# Julia Fundamental Types

## Array Indexing

- We've already seen the basics of array indexing. Let `x = [1, 2, 3, 4, 5]`

  What is `x[0]` ? `x[1]` ?

# Julia Fundamental Types

## Array Indexing

- We've already seen the basics of array indexing. Let `x = [1, 2, 3, 4, 5]`

  What is `x[0]` ? `x[1]` ?

  And `x[end-1]` ?

# Julia Fundamental Types

## Array Indexing

- We've already seen the basics of array indexing. Let `x = [1, 2, 3, 4, 5]`

  What is `x[0]` ? `x[1]` ?

  And `x[end-1]` ?

  And `x[1:3]` ?

# Julia Fundamental Types

## Array Indexing

- Same for 2D-arrays. Let `x = randn(2,2)`

# Julia Fundamental Types

## Array Indexing

- Same for 2D-arrays. Let `x = randn(2,2)`

  What is `x[1,1]` ?

# Julia Fundamental Types

## Array Indexing

- Same for 2D-arrays. Let `x = randn(2,2)`

  What is `x[1,1]` ?

  And `x[1,:]` ?

# Julia Fundamental Types

## Array Indexing

- Same for 2D-arrays. Let `x = randn(2,2)`

  What is `x[1,1]` ?

  And `x[1,:]` ?

  And `x[:,1]` ?

# Julia Fundamental Types

## Array Indexing

- Funny (but useless ?) : Booleans can be used to extract elements

# Julia Fundamental Types

## Array Indexing

- Funny (but useless ?) : Booleans can be used to extract elements

```
a = [1 2; 3 4]
```

# Julia Fundamental Types

## Array Indexing

- Funny (but useless ?) : Booleans can be used to extract elements

```
a = [1 2; 3 4]
b = [true false; false true]
```

# Julia Fundamental Types

## Array Indexing

- Funny (but useless ?) : Booleans can be used to extract elements

  ```
  a = [1 2; 3 4]
  b = [true false; false true]
  ```

  What is `a[b]` ?

# Julia Fundamental Types

## Array Indexing

- Less funny (but less useless) : some or all elements of an array can be set equal to one number using slice notation

# Julia Fundamental Types

## Array Indexing

- Less funny (but less useless) : some or all elements of an array can be set equal to one number using slice notation

```
a = zeros(4)
```

# Julia Fundamental Types

## Array Indexing

- Less funny (but less useless) : some or all elements of an array can be set equal to one number using slice notation

```
a = zeros(4)
a[2:4] .= 5
```

- What is `a` now ? And why do we need broadcasting ?

# Julia Fundamental Types

## Array Manipulation

- Always remember that a name of a variable / array points to data in RAM, it's **not** associated with it!

```
x = [1 2 3]

y = x

z = [2 3 4]

y = z
```

- What are x, y and z?

# Julia Fundamental Types

## Array Manipulation

- Always remember that a name of a variable / array points to data in RAM, it's **not** associated with it !

```
x = [1 2 3]
y = x
z = [2 3 4]
y = z
```

- What are x, y and z ?
- And now ?

```
x = [1 2 3]
y = x
z = [2 3 4]
y .= z
```

# Julia Fundamental Types

## Array Methods

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

- Let `a = [-1, 0, 1]`

```
@show length(a)
@show mean(a)
@show sum(a)
@show std(a)
@show var(a)
@show minimum(a)
@show maximum(a)
@show extrema(a)
```

# Julia Fundamental Types

## Array Methods

You can sort arrays :

- Try `b = sort(a, rev = true)`
- What is the difference with `b = sort!(a, rev = true)`

# Julia Fundamental Types

## Array Methods

We can test if they are identical and if they share the same memory

- Try `a == b`
- What is the difference with `a === b`?

# Julia Fundamental Types

## Array Methods

Broadcasting is crucial when working with arrays. Few examples :

- Compare `ones(2, 2) * ones(2, 2)` with `ones(2, 2) .* ones(2, 2)`
- Compare `ones(2, 2) + ones(2, 2)` with `ones(2, 2) .+ ones(2, 2)`
- Let `a = [10, 20, 30]` and `b = [0, 100, 100]`. What is `a .> b`? And `a .== b`?

# Julia Fundamental Types

## Tuples

- Julia has a built-in data structure called a tuple that is closely related to function arguments and return values
- A tuple is a fixed-length container that can hold any values, but cannot be modified (it is immutable)
- Tuples are constructed with commas and parentheses, and can be accessed via indexing :

# Julia Fundamental Types

## Tuples

```
t = (1.0, "test")
t[1] # access by index
a, b = t # unpack
t[1] = 3.0 # would fail as tuples are immutable
println("a = $a and b = $b")
```

# Julia Fundamental Types

## Named Tuples

As well as named tuples, which extend tuples with names for each argument

```
t = (α = 1.0, β = "test")
t.α # access by index
println("param 1 = $(t.α) and b = $(t.β)")
```

# Julia Fundamental Types

## Named Tuples

While immutable, it is possible to manipulate tuples and generate new ones

```
t2 = (γ = 4, δ = "test!!")
t3 = merge(t, t2) # new tuple
```