Tutorial_TrickyStuff

November 10, 2016

1 Tricky Stuff

This file highlights some tricky aspects of Julia (from the perspective of a matlab user)

Paul Söderlind (Paul.Soderlind at unisg.ch), April 2016, updated Nov 2016

```
In [1]: import Formatting                     #the first time, do Pkg.add("Formatting") to install the pack
        include("printmat.jl")                #just function for prettier matrix printing
```

```
Out[1]: printmat (generic function with 5 methods)
```

2 Julia Arrays Are Assigned by Reference

Julia arrays are designed to save memory. For instance, B=A means that both A and B point to the same memory addresses. (In contrast, MatLab creates a copy which doubles the memory requirement.)

2.1 Issue 1. B = A Creates Two Names of the *Same* Array

If A is an array, then after B=A, changing elements of A will modify B as well (and vice versa). In contrast, B = A + 0 (or B = A/2 or whatever) creates a completely new array which you can change without affecting A. If you really want a copy, do B = deepcopy(A).

Notice that this applies to arrays only. If A is a scalar or string, then B=A creates a copy. This means that you can later change A without affecting B.

The code below gives a few examples.

```
In [2]: A = [2 2]
        B = A
        C = sum(B)
        D = A + 0
        println("old A,B,C,D: ")
        printmat(A)
        printmat(B)
        printmat(C)
        printmat(D)
```

```
old A,B,C,D: 
         2         2

         2         2

         4

         2         2
```

```
In [3]: A[2] = -999
        println("after changing element A[2] to -999")
        println("new A,B,C,D: ")
        printmat(A)
        printmat(B)
        printmat(C)
        printmat(D)
```

```
after changing element A[2] to -999
new A,B,C,D:
         2      -999

         2      -999

         4

         2         2
```

2.2 Issue 2. Changing an Array Inside a Function Can Have Effects *Outside* the Function

When you use an array as a function argument, then that is passed as a reference to the function. This means that if you change some elements of the array (A[1] = A[1]/2, say) inside the function, then it will also affect the array outside the function (even if they have different names). In contrast, if you change the entire array (A/2, say) inside the function, then that does not affect the array outside the function.

Once again, this applies to arrays, but not to scalars or strings.

The code below defines a few different functions to illustrate this.

```
In [4]: function f1(A)
            A[1:end] = A[1:end]*10          #changes ELEMENTS of A
          return A
        end
        function f2(A)
            A = A*10                        #changes all of A
          return A
        end
        function f3(A)
            A = (A + 0)*10                  #changes all of A
          A[1:end] = A[1:end]*2
            #A = deepcopy(A)/2                #instead of the two previous lines, works too
          return A
        end

        x  = [1;2]
        x1 = deepcopy(x)                       #making copies
        x2 = deepcopy(x)
        x3 = deepcopy(x)
        println("original x (and x1, x2, x3 are the same): ")
        printmat(x)

        y1 = f1(x1)
        println("x1 (outside function) after calling f1(x1): ")
        printmat(x1)

        y2 = f2(x2)
        println("x2 (outside function) after calling f2(x2): ")
        printmat(x2)

        y3 = f3(x3)
        println("x3 (outside function) after calling f3(x3): ")
        printmat(x3)
```

```
original x (and x1, x2, x3 are the same):
         1
         2

x1 (outside function) after calling f1(x1):
        10
        20

x2 (outside function) after calling f2(x2):
         1
         2

x3 (outside function) after calling f3(x3):
         1
         2
```

Notice: when individual ELEMENTS of an array are changed inside a function, then this carries over to the array used in the function call. This is true also when we change all individual elements (as in f1()). It is not true when we work on the entire array (as in f2()) or change its shape. The solution to the problem with f1() is to do as in f3(): work on a copy of the input array.

2.3 Issue 3. A Reshaped Array still Refers to the Original Array

reshape() and vec() share the same memory addresses as the original arrays.

```
In [5]: x = [1 2;3 4]
        x1 = deepcopy(x)
        x2 = deepcopy(x)
        println("original x: ")
        printmat(x)

        y = reshape(x1,1,4)
        y[1] = -999
        println("x1 after changing y (where y = reshape(x,1,4): ")
        printmat(x1)

        y = vec(x2)
        y[1] = -27
        println("x2 after changing y (where y = vec(x): ")
        printmat(x2)
```

```
original x:
         1         2
         3         4

x1 after changing y (where y = reshape(x,1,4):
      -999         2
         3         4
```

```
x2 after changing y (where y = vec(x):
      -27         2
        3         4
```

3 A 1x1 Array Is Not a Scalar

and it often matters.

As an example, b = [1000] is a 1x1 array and cannot be used as a scalar. For instance, you cannot do A + b (if A is an array of a different size), or A[2] = b.

To use b as a scalar, just use b[1].

As another example, ones(2)'ones(2) creates an 1x1 array, but you can use it as a scalar by doing (ones(2)'ones(2))[1].

```
In [6]: A = [1 2]                                   #a 1x2 array
        b = [1000]                                  #a 1x1 array
        println("A and b: ")
        printmat(A)
        printmat(b)

        try
            y1 = A + b
        catch
            println("You cannot do A + b if A is a Txn array and b is a 1x1 array. Instead, do A + b[1]
            printmat(A + b[1])
            printmat(A .+ b)
        end

        try
            A[2] = b
        catch
            println("\nYou cannot do A[2] = b if A is a Txn array and b is a 1x1 array. Instead use A[2
            A[2] = b[1]
            printmat(A)
        end
```

```
A and b:
        1         2

     1000

You cannot do A + b if A is a Txn array and b is a 1x1 array. Instead, do A + b[1] or A .+ b
     1001      1002

     1001      1002


You cannot do A[2] = b if A is a Txn array and b is a 1x1 array. Instead use A[2] = b[1]
        1      1000
```

4 An Nx1 Array is not a Vector

and it sometimes matters.

Julia has both vectors and Nx1 arrays (the latter being a special case of NxM arrays). They can often be used interchangeably, but not always (as in the case of the dot() function below).

In particular, you typically use a vector when you want to pull out particular rows from a larger array.

```
In [7]: v = [1;2]                          #a vector with two elements
        v2 = (v')'                         #a 2x1 array

        println("v and v2 look similar, but they have different sizes: ")
        printmat(v)
        printmat(v2)
        println("size of v and v2: ",size(v)," ",size(v2))

        try
            println(dot(v2,v2))
        catch
            println("dot() requires vectors, so convert them by vec(v2): ",dot(vec(v2),vec(v2)))
        end


        x = [11 12;21 22;31 32]
        println("\nx, x[v,:] and x[v2,:]: ")
        printmat(x)
        printmat(x[v,:])
        printmat(x[v2,:])
```

```
v and v2 look similar, but they have different sizes:
         1
         2

         1
         2

size of v and v2: (2,) (2,1)
dot() requires vectors, so convert them by vec(v2): 5

x, x[v,:] and x[v2,:]:
        11        12
        21        22
        31        32

        11        12
        21        22

x[:,:,1]
        11
        21
x[:,:,2]
        12
        22
```

5 Creating Variables in a Loop

```
In [8]: for i = 1:5
          Tor = cos(i)
```

```
        end
        try
            println(Tor)
        catch
            println("Variables CREATED in a for loop are not visible outside the loop")
        end

        println("\nIn contrast, variables CHANGED in a for loop are visible outside the loop")
        Oden = Float64[]
        for i = 1:5
          Oden = cos(i)
        end
        println("Oden: ",round(Oden,4))
```

```
Variables CREATED in a for loop are not visible outside the loop

In contrast, variables CHANGED in a for loop are visible outside the loop
Oden: 0.2837
```

6 Adding Rows to an Array

```
In [9]: A =  [1 11]
        B =  [3 13]
        println("A and B")
        printmat(A)
        printmat(B)

        try
            A[2,:] = B
        catch
            println("\nTo append B at the end of A, you have to use [A;B].")
            printmat([A;B])
        end
```

```
A and B
         1        11

         3        13


To append B at the end of A, you have to use [A;B].
         1        11
         3        13
```

7 Cell Arrays

To creata a cell array, use Any[x1,x2,...]

Alternatively, you can preallocate as in B = Array{Any}(3) and then fill by, for instance, B[3] = 27

```
In [10]: A = Any[[11 12;21 22],"A nice dog",27]

         println("\nThe array A: ")
         for i = 1:length(A)
             printmat(A[i])
         end

         B = Array{Any}(3)
         B[1] = [11 12]
         B[2] = "A bad cat"
         B[3] = pi

         println("\nThe array B: ")
         for i = 1:length(B)
             printmat(B[i])
         end
```

```
The array A:
        11        12
        21        22

A nice dog

        27


The array B:
        11        12

A bad cat

π = 3.1415926535897...
```

8 New Things in Julia 0.5

```
In [11]: x = [11 12;21 22]
         println("x")
         printmat(x)

         println("x[1,:] gives a row vector in Julia 0.4.x, but a column vector in Julia 0.5.x: ")
         printmat(x[1,:])

         println("do x[1:1,:] or x[[true;false],:] to get a row vector in either version: ")
         printmat(x[1:1,:])
         printmat(x[[true;false],:])
```

```
x
        11        12
        21        22

x[1,:] gives a row vector in Julia 0.4.x, but a column vector in Julia 0.5.x:
        11
        12

do x[1:1,:] or x[[true;false],:] to get a row vector in either version:
        11        12

        11        12
```

```
In [ ]:
```