ArrayIteratorInterfaces

May 16, 2018

0.1 The Array and Iterator Interface

Julia has a few "informal" interfaces. The idea is that, if you subclass a given abstract type and implement a few methods, then by multiple dispatch the type will now "act like" whatever you want it to act like, in any instance you would like.

One example is the array interface. The methods to implement are:

- `size(A)`
- `getindex(A, i::Int)`
- `getindex(A, I::Vararg{Int, N})`
- `setindex!(A, v, i::Int)`
- `setindex!(A, v, I::Vararg{Int, N})`

`size(A)` returns what the size of the array is as a tuple. For example, a 3-dimensional "array" (remember, it can be anything, we are just saying it acts like an array!) would have `size(A)` return a tuple `(dim1,dim2,dim3)` for the sizes along each dimension. `getindex` is the function that is used by `A[i]`, and `setindex!` is the function for mutating `A` via `A[i]=`.

The `Vararg{Int, N}` just means variable numbers of arguments, so for example `getindex(A,i1,i2)` is what is called by `A[i1,i2]`. More generally, using slurping you can define `getindex(A,I...)` and the function for `A[i1,i2,...,in]`, and `I` will be a tuple of `(i1,i2,...,in)`.

Reference: http://docs.julialang.org/en/release-0.5/manual/interfaces/

0.2 Usage Example

One usage example is in DifferentialEquations.jl. When you solve a differential equation, the solution returns a specialized solution type.

```
In [5]: using DifferentialEquations, DiffEqProblemLibrary
        sol = solve(prob_ode_linear)
```

```
Out[5]: retcode: Success
        Interpolation: 3rd order Hermite
        t: 5-element Array{Float64,1}:
         0.0
         0.0996426
         0.345703
         0.677692
```

```
 1.0
u: 5-element Array{Float64,1}:
 0.5
 0.552939
 0.708938
 0.99136
 1.3728
```

While there are many fields that the type could hold, such as the errors of a given analytical solution:

```
In [3]: println(sol)
```

```
retcode: Success
Interpolation: 3rd order Hermite
t: [0.0, 0.0996426, 0.345703, 0.677692, 1.0]
u: [0.5, 0.552939, 0.708938, 0.99136, 1.3728]
```

An array interface is provided so that the solution "acts" like a traditional solution: i.e. as an array of the numerically computed values:

```
In [4]: println(sol.t[3]) # The third timestep
        sol[3] # The value at the third timestep
```

```
0.3457030604980719
```

```
Out[4]: 0.7089380797962724
```

An entire Julia organization, JuliaArrays, is devoted to making fast, interesting, and easy to use arrays. It's an odd concept: arrays are just arrays, right? However, Julia's genericism makes it so that way this now works in any place which accepts an `AbstractVector` output, so the solution is directly usable in numerical optimization algorithms, regressions, etc.