Julia Introduction[1]

Introduction

Julia is a new programming language that runs much faster than Python; it is possible to write fast code in MATLAB where everything is written as linear algebra or using carefully optimized numpy or fancy just-in-time compilation in Python; the idea of Julia is that it runs at C-like speeds for natural, modern-looking code. It does this by allowing, while not requiring, type declaration and by not having classes, instead it has *types*, a bit like *structs* in C, and multiple dispatch.

It has other features to make it useful for scientific computing, little things like being able to `2v` when you mean `2*v`, along with big things, like a sophisticated multi-dimensional array datatype that can be used for efficient matrix operations. Presumably to help persuade people to finally abandon MATLAB it has a MATLAB-like syntax, blocks are denoated using a `end` keyword, the first element of an array `a` is `a[1]` and `1:10` means one to ten, not one to nine.

If you are used to Python Julia can seem frustrating to debug, mostly because the typing can be hard to get used to, but debugging Julia as a Python programmer really reminds you how often you cast variables without even noticing; this is part of why Julia is much faster.

This only outlines the simplest parts of the language, the wikibook

> https://en.wikibooks.org/wiki/Introducing_Julia/

is a good place to look for a longer introduction. There is online Julia at

> https://juliabox.com/

A simple example

Here is a programme to add powers of two (`add.jl`):

```
1  highest_power=10
2
3  value=1.0::Float64
4  current=0.5::Float64
5
6  for i in 1:highest_power
7     value+=current
8     current*=0.5
9  end
10
11 println(value)
```

Line 1 defined `highest_power`; this is dynamically typed, as in Python, but `value` and `current` are given a type, `Float64`; as an indication of how seriously it takes typing, is **line 4** was `value=1::Float64` it would return an error since `1` isn't a `Float64`. You can find a full list of types in the wikibook, it has lots of different int and float types, along with rational numbers using `//` to seperate numerator and divisor (`rational.jl`):

[1] `https://github.com/conorhoughton/teaching_misc/tree/master/python_workshop/`

```
1 a=2//3
2 b=1//2
3 println(a-b)
```

Arrays

Arrays are what Python calls lists, python is the odd one out here, array is a more common name. Julia has the same slicing functionality as Python, although as mentioned above indexing is different (`slice.jl`)

```
1 a=[1,2,3,4,5]
2 println(a[1:3])
3
4 for i in a
5    println(i)
6 end
```

prints `[1,2,3]` from **line 2**, **line 4** to **line 6** demonstrates a for loop. The last element in an array is indexed `end` so in the programme above `a[end]` is `5`.

Arrays can store mixed items, but the array can be typed, so `a` in (`typed_list.jl`

```
1 a=Int64[1,2,3,4,5]
2 push!(a,6)
3 println(a)
```

can only store items of type `Int64`. `push!` pushes an item onto the list, like `append` in Python, again, this is the more common notation. The `!` is part of an convention where all commands that change an array have a `!`.

As mentioned above, Julia arrays can be multidimensional and have matrix like operations, but this won't be explored in this brief overview. There is also a tuple type which is immutable.

Functions

Here is a programme with some functions (`functions.jl`)

```
1  function add_to_int(a::Integer,b::Integer)
2           println("int version")
3           a+b
4  end
5
6  function add_to_int(a::Real,b::Real)
7           println("float version")
8           convert(Int64,a+b)
9  end
10
11 function add_to_int(a,b)
12          println("what are these things")
13          0
14 end
15
```

```
16 println(add_to_int(12,6))
17 println(add_to_int(12.0,6.0))
```

Obviously this is a very artificial example, but it shows some of the features of functions, first, their return value is the most recently evaluated expression and second, and more importantly, they support multiple dispach; the function is chosen to match the type of the arguments, here there is one function for `Integers`, this is a supertype which includes, for example, `Int64`, there is one for `Real`, the supertype that includes various floats, and one with no type; the correct function is used for each. If there is no correct fuction there will be an error.

Composite Types

Julia doesn't have classes, this comes as a surprise at first, but it does have composite data types that work like structs, combining this with multiple dispach captures important parts of the functionality of classes, in a way that supports fast code (`struct_example.jl`

```
1  mutable struct Cow
2          name::String
3          age::Int64
4  end
5
6  mutable struct Poem
7          name::String
8  end
9
10 function move(cow::Cow)
11          print(cow.name," walks forward")
12          println("showing the weight of her ",cow.age," years")
13 end
14
15 function move(poem::Poem)
16          println(poem.name, " moves us to tears with its beauty")
17 end
18
19 poem = Poem("The Red Wheelbarrow")
20 cow = Cow("Hellcow",42)
21
22 move(cow)
23 move(poem)
```

You can see that although the structs have no methods, the function `update` can have different meaning for the two different data types. The default constructor defines the variables in the order they appear, it is possible to define other constructors, but that won't be considered here.

An exercise

Write a short Julia programme to implement an leaky integrate-and-fire neuron. The leaky integrate-and-fire neuron is the simplest common model of a spiking neuron. In the model the

volage in the neuron satisfies

$$\tau_m \frac{dv}{dt} = E_l - v + V_I \tag{1}$$

where τ_m and E_l are parameters, and I is some input, in fact it is a voltage derived from an input current into the cell; $V_I = IR_m$ where I is a current and R_m is the member resistance, a property of the cell membrane. Now neurons spike, send out pulses of voltage, to include this in the simple model the differential equation is supplemented with the rule that if the voltage exceeds some threshold value $v > V_t$ then it is reset to a reset value $v = V_r$.

The idea here is to implement this, using a `struct` for the neuron and evolving the neuron using either Euler's method

$$v \to v + \frac{dv}{dt}\delta t \tag{2}$$

or fourth-order Runge-Kutta, which you can find online. Good values for the constants would be $\tau_m = 10\,\text{ms}$, $V_t = -55\,\text{mV}$, $V_r = E_l = -77\,\text{mV}$ and $V_I = 23\,\text{mV}$. A time step of $\delta t = 0.1\,\text{ms}$ would work. I tried something like this and found Julia was forty times faster than Python.