Introduction to Julia for Python developers

PyData, Berlin
May, 2016

David Higgins

Bernstein Center for
Computational Neuroscience
Berlin

Technische
Universität
Berlin

Who is this guy?

Have been programming and involved in Open Source since the 1980’s

Currently working as a computational neuroscientist
-requires both rapid prototyping and scalability

I tend to work on projects which are highly numerical, requiring both precision and processing power

I’ve been using Julia in my daily work for 2 years now

And I teach a course on Advanced Scientific Programming using Python at the Humboldt and Technical universities

What is Julia?

It is a new programming language (begun 2009, first release 2012)

Produced by a group at MIT with a long history of developing tools for parallelising existing programming languages

It is a General Purpose language with Numerical Computing at the core of its design

Who should be interested in Julia?

Data scientists, anyone interested in performance in numerical computing

If you are already pushing the bounds of Python via Cython

If you want a faster route from prototype to high-performance implementation

If you want a new toy

Why *you* should be interested in Julia

Syntactically it is easy to learn from Python

All Python libraries are fully usable via PyCall command

The development cycle is similar to Python (rapid prototyping)

The code runs at C-like speeds

Parallelisation is an integral part of the language

2-language problem

Non-application based development is typically interactive nowadays

Python has shown us how a dynamically typed, interpreted language can be easily used to solve our problems

Rapid prototyping and low barrier to entry are important language characteristics

But what happens when you need to scale things up?

2-language problem

Typical solutions involve using 2 languages

Python, matlab, mathematica, etc. for investigatory work and prototype development

C, Fortran, specialised solutions (or even just recoding in Cython) for performance implementations

This is a huge development overhead

Technical description

Julia

- is dynamically typed, but with optional types
- built-in types are equivalent to user-defined types
- is JIT compiled using LLVM
- utilises dynamic multiple dispatch
- has full metaprogramming capabilities
- can call C and Fortran libraries natively

Dynamical language with optional Typing

Python is dynamically typed

C is statically typed

Knowing the type of a variable increases the speed of the compiled program

Needing to annotate your code with types slows down the coding process

So in Julia it’s optional

Dynamical language with optional Typing

Use of types is largely *not* required to obtain the performance boosts!

It will only help slightly

But *multiple dispatch* largely removes the overheads of type checking which exist in Python

It's mainly useful for ensuring that the correct code gets called in the right place

Dynamical language with optional Typing

Additional comments on types:

Despite variables having different types, promotion to compatible types automatically occurs

eg. >>1 + 0.234   Int + Float = ?

Most importantly, no meaningful difference between built-in data types (eg. Int64, Float64) and user-defined types

LLVM at the core

LLVM is a compiler infrastructure project

Julia generates LLVM Intermediate Representation (IR) code

LLVM takes this and optimises it, then compiles it for the host system

Finally the machine representation executes

This approach, apart from being fast, allows for inspection of the intermediate code

Multiple dispatch

Overloaded functions with late binding…

Class based OOP = dynamic single dispatch

```
A.function_call()
```

Function overloading (C++) = static multiple dispatch

```
set_parameter(A, 3.0)
```

Julia methods are defined and called based on acceptable parameter data types = dynamic multiple dispatch

```
my_func(A,B,C,22.3)
```

This feature is where much of the speed comes from!

Metaprogramming

Macros are applied at code interpretation-time

This allows for automated code generation (which makes your code cleaner to read)

And also, optimisations for the actual number of parameters or values used are possible (eg. explicit calculation of mathematical equations rather than generic solutions)

C and Fortran can be called ‘natively’

Many numerical libraries are written in C or Fortran

Julia can call these library functions directly:

```
ccall(:clock, Int32, ())
```

The JIT instructions generated are identical to those generated by a C compiler, no overhead!

Must be from a shared library

You are responsible for Type compatibility!

Getting Julia

http://julialang.org

http://docs.julialang.org

https://groups.google.com/forum/#!forum/julia-users

Interacting with Julia

Command-line invocation

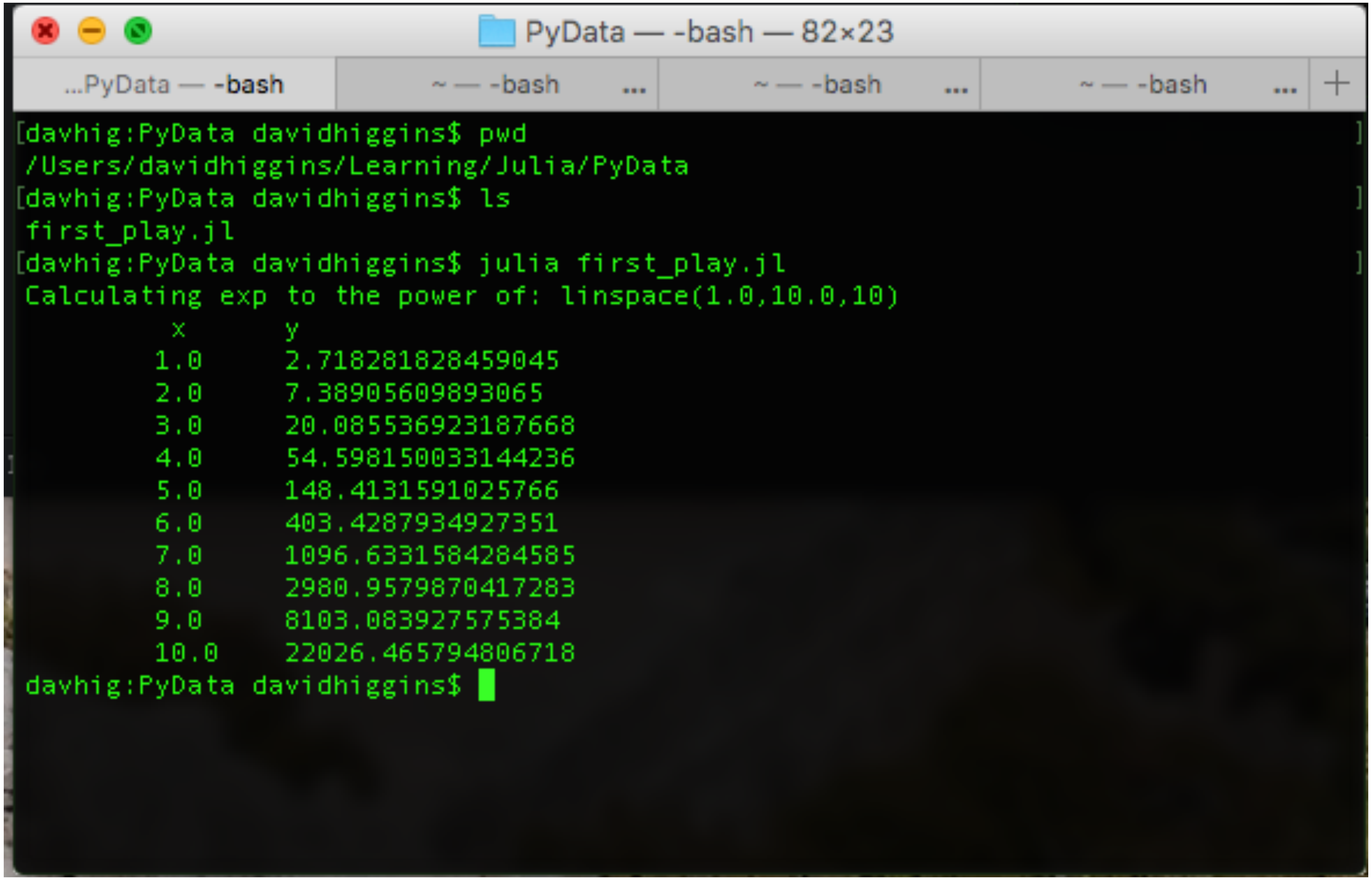
REPL

Jupyter (IPython like notebooks)

Juno + Atom (formerly with LightTable)

Interacting with Julia

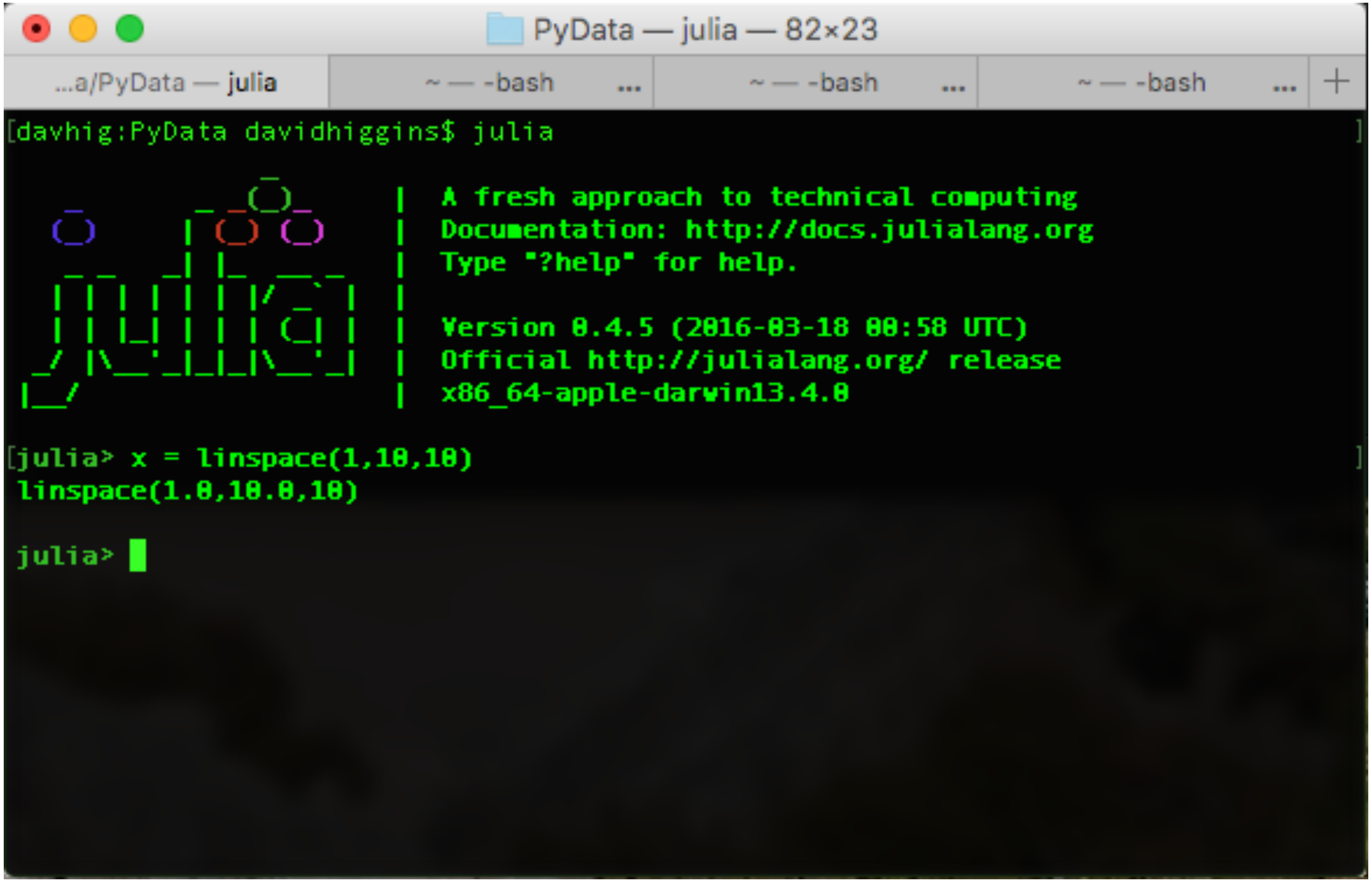
Command-line invocation



```
davhig:PyData davidhiggins$ pwd
/Users/davidhiggins/Learning/Julia/PyData
davhig:PyData davidhiggins$ ls
first_play.jl
davhig:PyData davidhiggins$ julia first_play.jl
Calculating exp to the power of: linspace(1.0,10.0,10)
         x      y
        1.0     2.718281828459045
        2.0     7.38905609893065
        3.0     20.085536923187668
        4.0     54.598150033144236
        5.0     148.4131591025766
        6.0     403.4287934927351
        7.0     1096.6331584284585
        8.0     2980.9579870417283
        9.0     8103.083927575384
        10.0    22026.465794806718
davhig:PyData davidhiggins$
```

Interacting with Julia

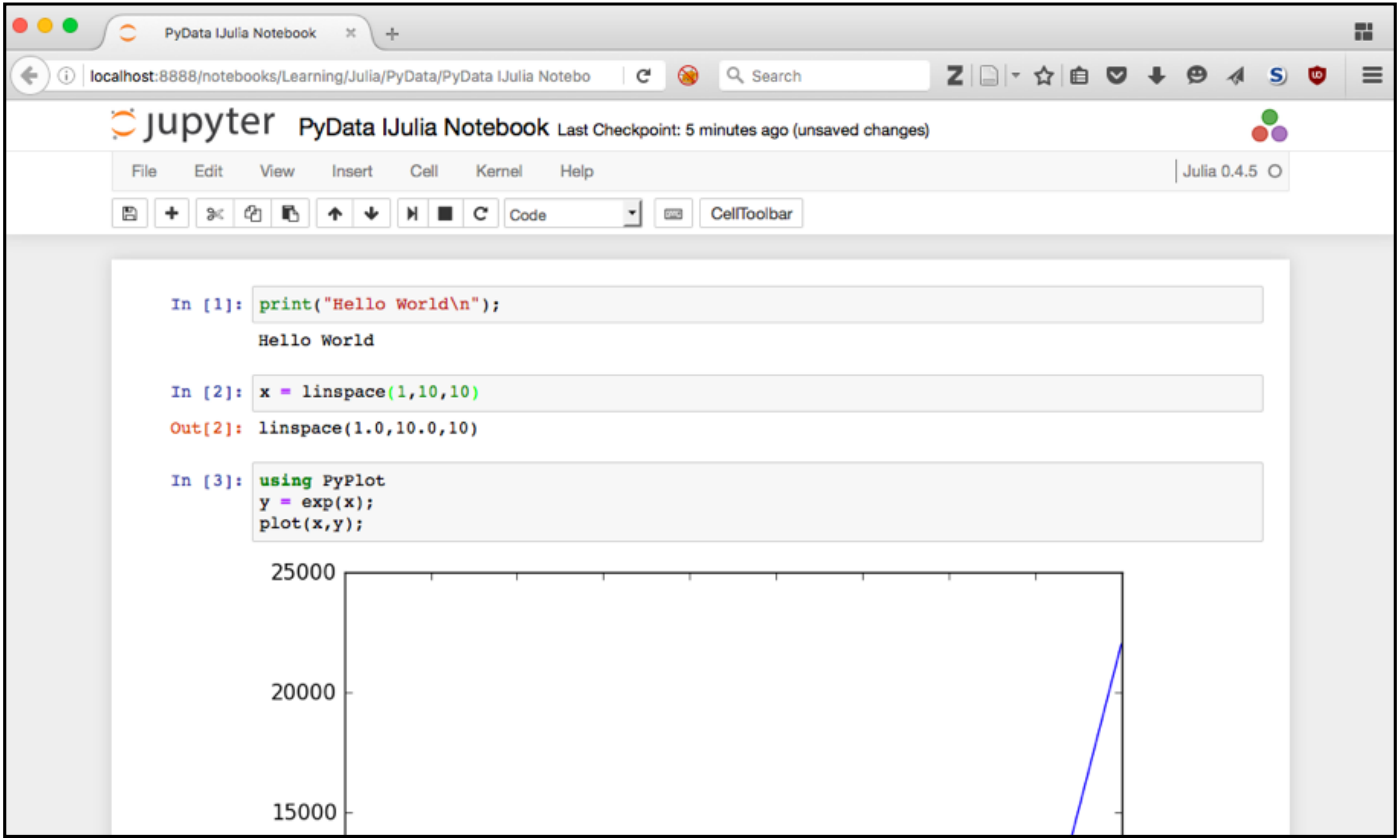
Read-Eval-Print-Loop (REPL)



```
[davhig:PyData davidhiggins$ julia
               _
   _       _ _(_)_     |  A fresh approach to technical computing
  (_)     | (_) (_)    |  Documentation: http://docs.julialang.org
   _ _   _| |_  __ _   |  Type "?help" for help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 0.4.5 (2016-03-18 00:58 UTC)
 _/ |\__'_|_|_|\__'_|  |  Official http://julialang.org/ release
|__/                   |  x86_64-apple-darwin13.4.0

[julia> x = linspace(1,10,10)
linspace(1.0,10.0,10)

julia> 
```

Interacting with Julia

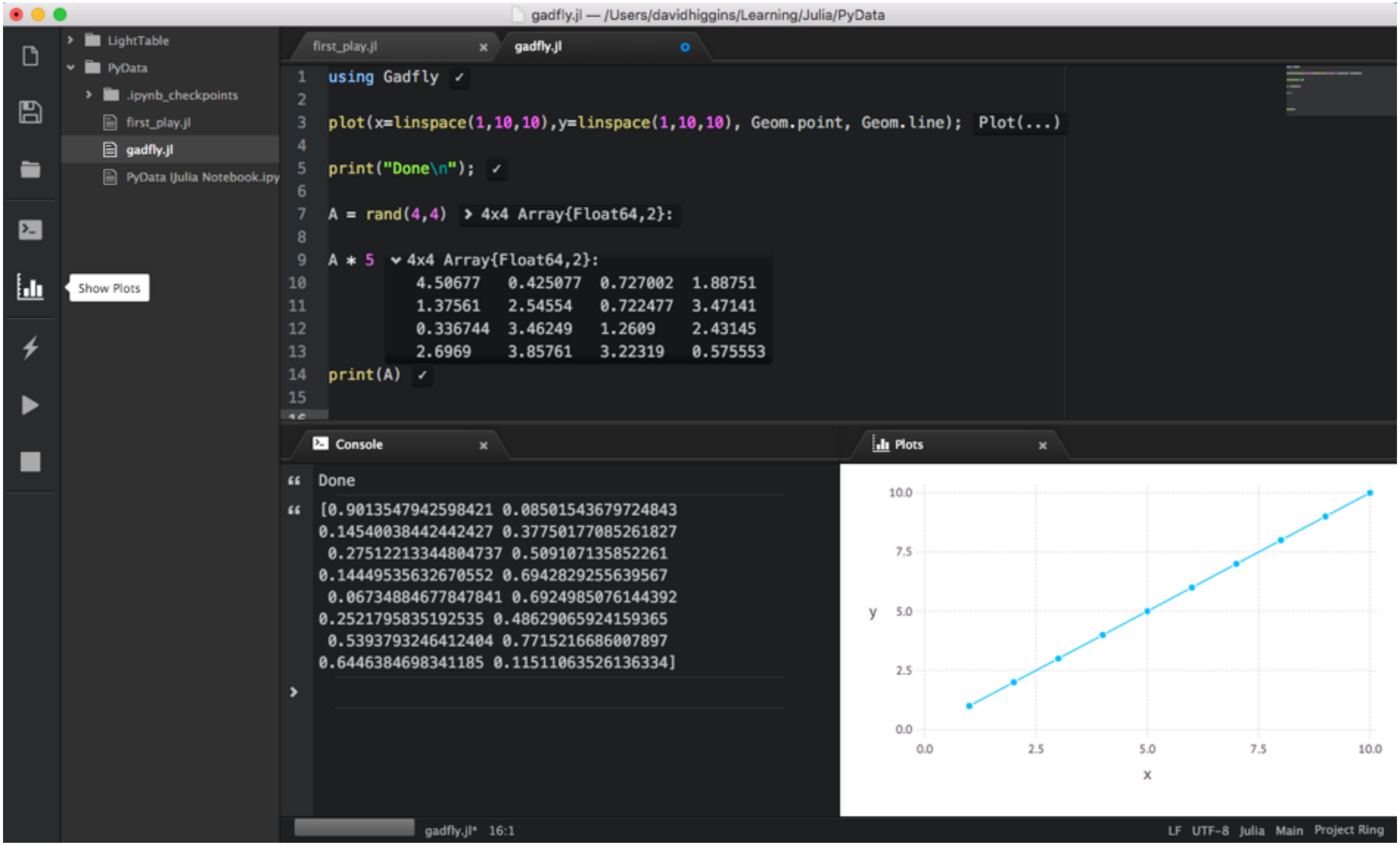
Jupyter Notebooks (IJulia)


PyData IJulia Notebook
localhost:8888/notebooks/Learning/Julia/PyData/PyData IJulia Notebo
Search
jupyter
PyData IJulia Notebook Last Checkpoint: 5 minutes ago (unsaved changes)
File
Edit
View
Insert
Cell
Kernel
Help
Julia 0.4.5
Code
CellToolbar
25000
20000
15000

```
In [1]: print("Hello World\n");

        Hello World

In [2]: x = linspace(1,10,10)

Out[2]: linspace(1.0,10.0,10)

In [3]: using PyPlot
        y = exp(x);
        plot(x,y);
```

Interacting with Julia

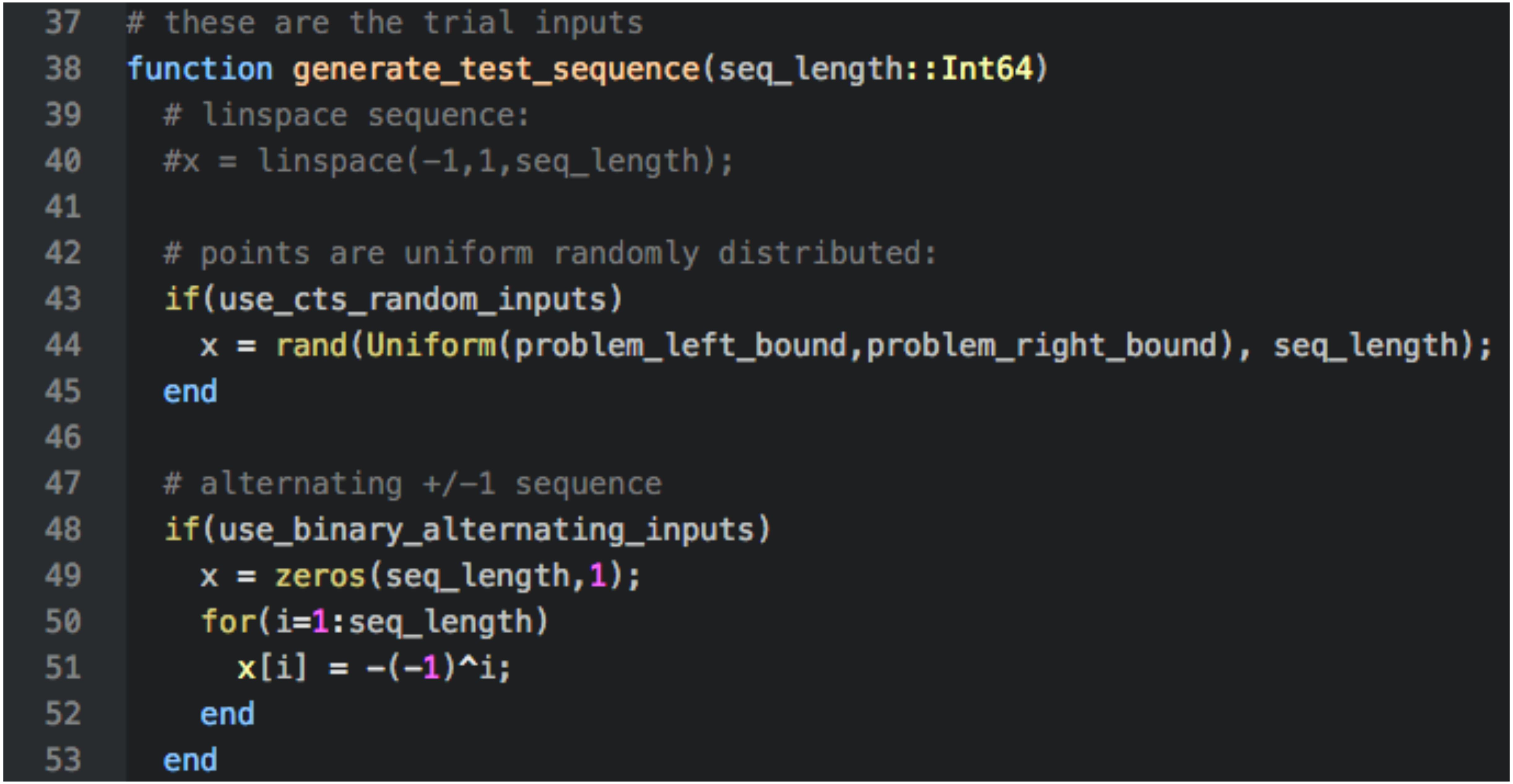
Atom + Juno (formerly used LightTable)



```
using Gadfly ✓

plot(x=linspace(1,10,10),y=linspace(1,10,10), Geom.point, Geom.line); Plot(...)

print("Done\n"); ✓

A = rand(4,4) > 4x4 Array{Float64,2}:

A * 5  ⌄ 4x4 Array{Float64,2}:
          4.50677   0.425077  0.727002  1.88751
          1.37561   2.54554   0.722477  3.47141
          0.336744  3.46249   1.2609    2.43145
          2.6969    3.85761   3.22319   0.575553
print(A) ✓
```

```
Done
[0.9013547942598421 0.08501543679724843
0.14540038442442427 0.37750177085261827
 0.27512213344804737 0.509107135852261
0.14449535632670552 0.6942829255639567
 0.06734884677847841 0.6924985076144392
0.2521795835192535 0.48629065924159365
 0.5393793246412404 0.7715216686007897
0.6446384698341185 0.11511063526136334]
>
```

Forget about indentation



```
# these are the trial inputs
function generate_test_sequence(seq_length::Int64)
  # linspace sequence:
  #x = linspace(-1,1,seq_length);

  # points are uniform randomly distributed:
  if(use_cts_random_inputs)
    x = rand(Uniform(problem_left_bound,problem_right_bound), seq_length);
  end

  # alternating +/-1 sequence
  if(use_binary_alternating_inputs)
    x = zeros(seq_length,1);
    for(i=1:seq_length)
      x[i] = -(-1)^i;
    end
  end
```

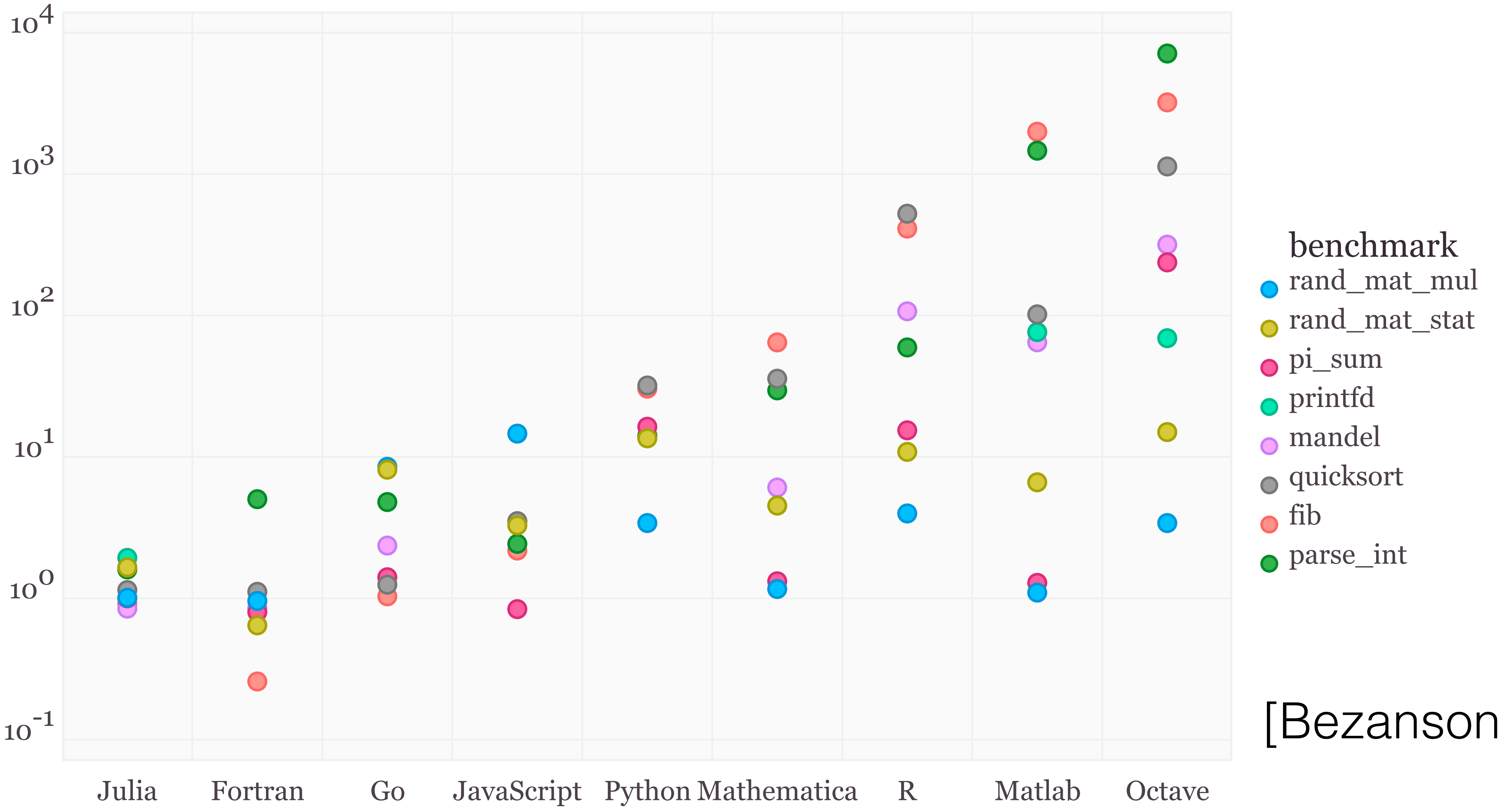
Basic syntax

Switch to Jupyter notebook

Julia in the numbers

493 contributors (this week)

966 registered packages



[Bezanson; ArXiV]

Summary

Julia is

- quick to write
- fast to run

- a general purpose programming language
- designed from the ground up for numerical computing
- with parallel processing
- and easy scalability

- all of your existing tools still work in it

Thank you

Organisers

Audience

Language designers and developers

http://julialang.org

http://docs.julialang.org

https://groups.google.com/forum/#!forum/julia-users