# Github Badges

[Shields.io: Quality metadata badges for open source projects](https://shields.io/)

# @edit

In Julia REPL, what ever command you run, if you prefix it with @edit, Julia will show its code, as its done

Try @edit 2+2

# @which

In Julia, we know 2+2 is 4, but for Julia operator means differently, for example

You can do 2+2, also image+image / or URL+URL/ so to know, what logic Julia is applying, you can just run that operator, and Julia will tell you how many function are assigned to that operator like below for +

Graphical user interface, text, application

Description automatically generated

And now when you run anything, prefix that with @which, Julia will tell you what function was used

A picture containing text

Description automatically generated

And you can click on it, and Julia will take you to that function

You can also list all the methods , assigned to a operator

Text

Description automatically generated

# [Adding Julia to PATH on Windows 10,](https://julialang.org/downloads/platform/#adding_julia_to_path_on_windows_10)

to run from command prompt

1. Open Run (Windows Key + R), type in rundll32 sysdm.cpl,EditEnvironmentVariables and hit enter.
2. Under either the "User Variables" or "System Variables" section, find the row with "Path", and click edit.
3. The "Edit environment variable" UI will appear. Here, click "New", and paste in the directory noted from the installation stage. This should look something like (verify if changed) C:\Users\ankit\AppData\Local\Programs\Julia\Julia-1.4.1\bin
4. Click OK. You can now run Julia from the command line, by typing julia!

# Julia in Visual Studio Code

For opening up REPL in VSC

A screenshot of a computer

Description automatically generated with medium confidence

CNTL + SHIFT +P 🡪 REPL start by Alt + J

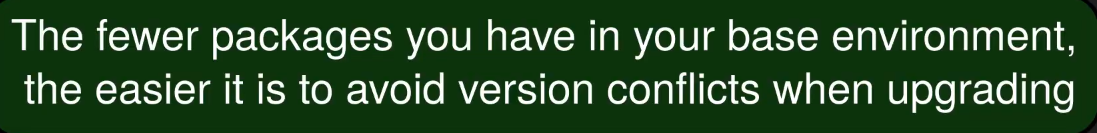
A screenshot of a computer screen

Description automatically generated with medium confidence

St command in pkg mode, will give everything added in your current environment

Text

Description automatically generated



Use rm command and name of package to remove

# Activate Environment in Julia

List of environments - (@v1.4) default

(Test\_Env) – creating this as test ground

# Creating project template

So what needs to be done for creating a new Project:

1. In VSCODE go to this location and open mytemplate.jl

D:\MyGit\MyJulia (and compile it)

Text

Description automatically generated

1. Once done, use the command below (in REPL) with new project name and see the new directory being created

Text

Description automatically generated

1. You will see new project dirs. Created in MYGIT dir

Graphical user interface, text, application

Description automatically generated

1. Open Project in VSCode, see the bottom ribbon, environment should not be Julia 1.4 but project name instead

Text

Description automatically generated

1. See

Graphical user interface, text, application, chat or text message

Description automatically generated

Graphical user interface

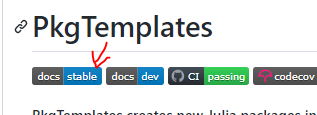
Description automatically generated with low confidence

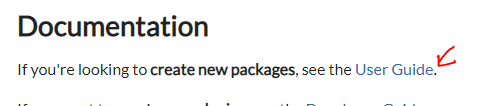
1. Now connect and synch with Github (issue git remote -v), create exactly the same repo with even .jl added to it, come back here and push, (git push origin master)

Text

Description automatically generated

Go here [invenia/PkgTemplates.jl: Create new Julia packages, the easy way (github.com)](https://github.com/invenia/PkgTemplates.jl)





Graphical user interface, text, application, email

Description automatically generated

Template(;

user="my-username",

dir="~/code",

authors="Acme Corp",

julia=v"1.1",

plugins=[

License(; name="MPL"),

Git(; manifest=true, ssh=true),

GitHubActions(; x86=true),

Codecov(),

Documenter{GitHubActions}(),

Develop(),

],

)

I used below template (D:\MyGit\MyJulia)

using PkgTemplates

t = Template(;

user="ankit48365", # GitHub username

dir="D:\\MyGit", # directory where new project will be created

authors="ankit DS",

julia=v"1.4.1",

plugins=[

License(; name="MIT"),

Git(; name="ankit48365", email="ankit48365@gmail.com", manifest=true), # store Manifest.toml in git

# GitHubActions(; linux=true, windows=false, osx=false), # use GitHub Actions for CI testing

# Documenter{GitHubActions}(), # build documentation with GitHub Actions

# Codecov(), # use CodeCov for test coverage information

# Develop(), # add the dev'ed package to the current environment

#BlueStyleBadge(), # follow the BlueStyle coding guidelines

#ColPracBadge(), # follow the ColPrac contribution guidelines

],

)

# t("MyNewProject") - this is how you will initiate a new project (T is the function you created above)

# Small Things in Julia

* D
* F
* G
* H
* A
* S
* F
* T
* G
* Today() – reveals the date (mention using dates)
* versioninfo() the version details
* Pkg.status() , all the packages and there details (mention using Pkg)
* Double equal vs triple equal (== vs ===)

Below a==b will be true, as both hold same elements and same structure, but a===b will give false as memory stoage of a and b are diffrent

julia> a = [1,2,3,4]

4-element Array{Int64,1}:

1

2

3

4

julia> b = Int.(a)

4-element Array{Int64,1}:

1

2

3

4

* Arrays are indexed with square brackets (a[1]), with indexing starting at 1 by default
* Vd

# Build Project / Portfolio

<https://towardsdatascience.com/develop-and-sell-a-python-api-from-start-to-end-tutorial-9a038e433966>

<https://towardsdatascience.com/a-complete-data-science-portfolio-project-ebbced35ea84>

# Pluto

<https://www.youtube.com/watch?v=IAF8DjrQSSk&feature=emb_rel_end>

for reactive interactive notebook

Text

Description automatically generated

Text

Description automatically generated

Pluto.run() to run it then

# Julia Tech stack

For my tech-stack, I will be using:

* DataFrames.jl data frames,
* Gadfly.jl for visualization, and
* Lathe.jl for statistical analysis and machine-learning

Another Guy used this

* Github (Code hosting),
* Anaconda (Dependency and environment management),
* Jupyter Notebook (code development and documentation),
* Python (programming language),
* AWS (deployment),
* Rapidapi (market to sell)

# Deep Learning

Deep learning Frame Work for All Languages – **Mocha** is for Julia

<https://www.datasciencecentral.com/profiles/blogs/deep-learning-libraries-by-language>

[Mocha](https://github.com/pluskid/Mocha.jl) is a Deep Learning framework for Julia, inspired by the C++ framework Caffe.

# Flux

<https://fluxml.ai/>

<https://github.com/FluxML>

<https://github.com/FluxML/model-zoo>

<https://fluxml.ai/Flux.jl/stable/ecosystem/>

# Genie

<https://genieframework.github.io/Genie.jl/> - all reading material is here

# Good Julia Sources

<https://julialang.org/learning/>

<https://exercism.io/my/tracks/julia> - github login

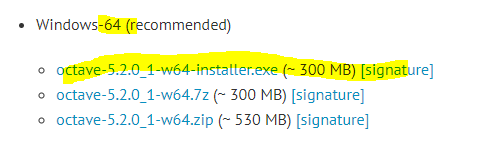
# Matlab

<https://lakras.github.io/matlab-to-julia/>

MATLAB is not free, so GNU Octave is a free open source, all the MATLAB code runs here

<https://www.gnu.org/software/octave/>

<https://www.gnu.org/software/octave/#install>



# iJulia

using Pkg

Pkg.add("IJulia")

Using IJulia

notebook()

# if issues then -🡪 Pkg.build("IJulia")

julia> ;jupyter notebook someJuliaNotebook.ipynb 2>/dev/null &

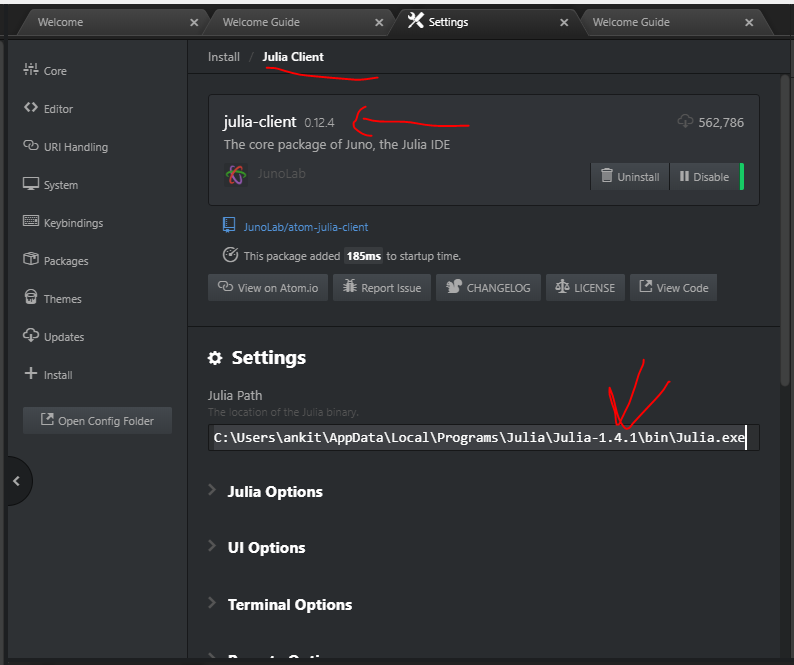
# Julia Path in Atom

C:\Users\ankit\AppData\Local\Programs\Julia\Julia-1.4.1\bin\Julia.exe

C:\Users\ankit\.julia\packages\JuliaAcademyData\1to3l\courses\Foundations of machine learning

/srv/julia/pkg/packages/JuliaAcademyData/1to3l/courses/World of machine learning/visualize/demosetup.j

C:/Users/yourUserID/.julia/packages/JuliaAcademyData/1to3l/courses/World of machine learning/visualize/demosetup.jl



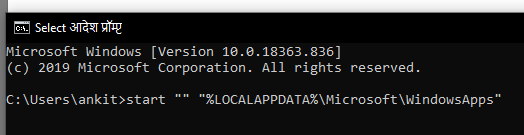
# Excercism

<https://github.com/exercism/cli/releases/tag/v3.0.13>



On CMD then do….

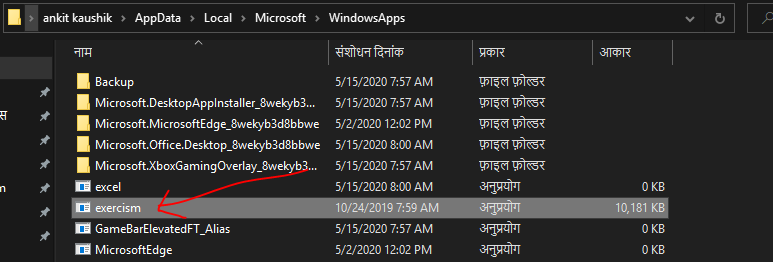
start "" "%LOCALAPPDATA%\Microsoft\WindowsApps"



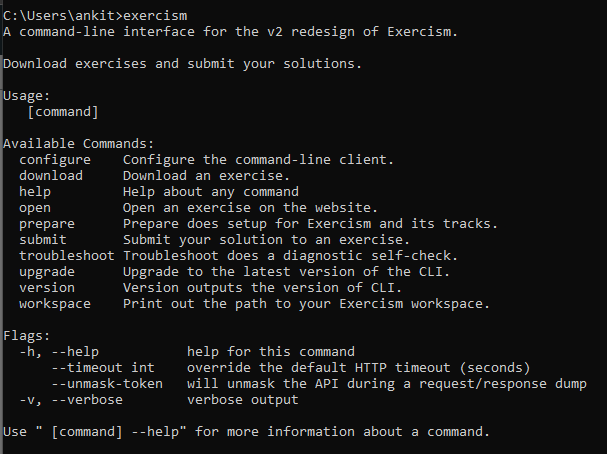
Hit and enter and below link will open

C:\Users\ankit\AppData\Local\Microsoft\WindowsApps

Place the exe file

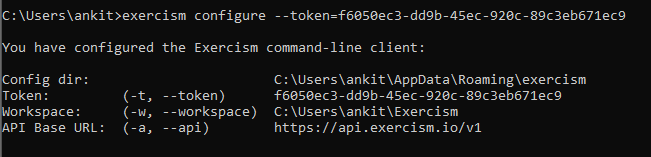


Close CMD and reopen again and issue below command, if it opens like below, that means successful.

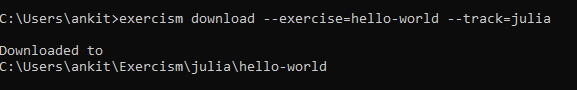


Then configure

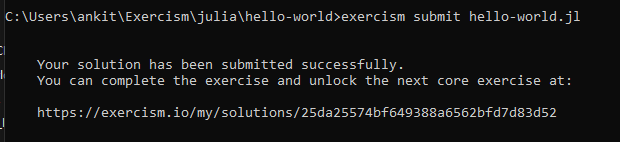
exercism configure --token=f6050ec3-dd9b-45ec-920c-89c3eb671ec9

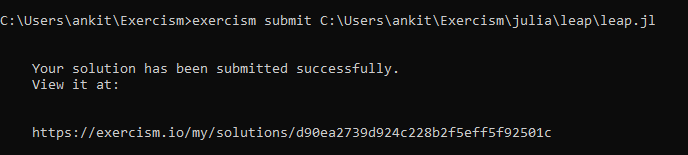


Download exercise like below



Submit like below





# Julia Web Development

Julia’s own website is made with [tlienart/Franklin.jl](https://github.com/tlienart/Franklin.jl" \t "_blank" \o "github.com) a Julia package. (good for static pages)

[GenieFramework/Genie.jl](https://github.com/GenieFramework/Genie.jl) is probably your best bet for a dynamic website and I guess you could use those together.

\*\* Django/Python should also be usable with Julia, and I would like to try or see anyone do it…

\*\* e.g [JuliaGizmos/WebIO.jl](https://github.com/JuliaGizmos/WebIO.jl" \t "_blank" \o "github.com) “a bridge between Julia and web technologies” may be the newest. I stripped out a lot of text from the old answer below and added a new one above.

Not only can you make websites, you can make interactive UIs on the web in pure Julia at a very high level—an early example was Minesweeper code in 70 lines for the web

https://github.com/JuliaGizmos/Interact.jl#as-a-standalone-web-page

As a standalone web page

Any Julia function that returns an Interact-renderable object (such as a widget or the output of an @manipulate) can be repurposed to run as a simple web page served by the Mux web app framework.

using Mux, WebIO

function app(req) # req is a Mux request dictionary

...

end

webio\_serve(page("/", app), 8000) # this will serve at http://localhost:8000/

fastest way to deploy code on the cloud

https://www.youtube.com/watch?v=qycU7jwZ1fo

# Create a Package/ Document in Julia

[How to Create Software Packages with Julia Language - YouTube](https://www.youtube.com/watch?v=KdfFN02PuFo)

1. Create project using the template I have created to create projects, above in this document

Template saved here - D:\MyGit\MyJulia to call it, call function ‘akproj’ – run mytemplate.jl once

Text

Description automatically generated

1. Later run function akproj as shown above with new project name
2. Then in visual studio, close folder, or also close editor, and then file 🡪 open folder, and select the folder to open in new created project
3. At the bottom, environment would be project environment



1. Ran git remote -v 🡪 based on the command, created repo in github and then later ran *github push origin* to synch this git repo to github repo
2. Add document and documenters tools as below

Text

Description automatically generated

1. Now using documenter tool to generate

Graphical user interface, text, application

Description automatically generated

1. Once complied, run the command to generate

(tip: main JL file should have the same name as the package name or the project name, if env name is Ankit then you should have Ankit.jl under /src and it will document that only)

A screenshot of a computer

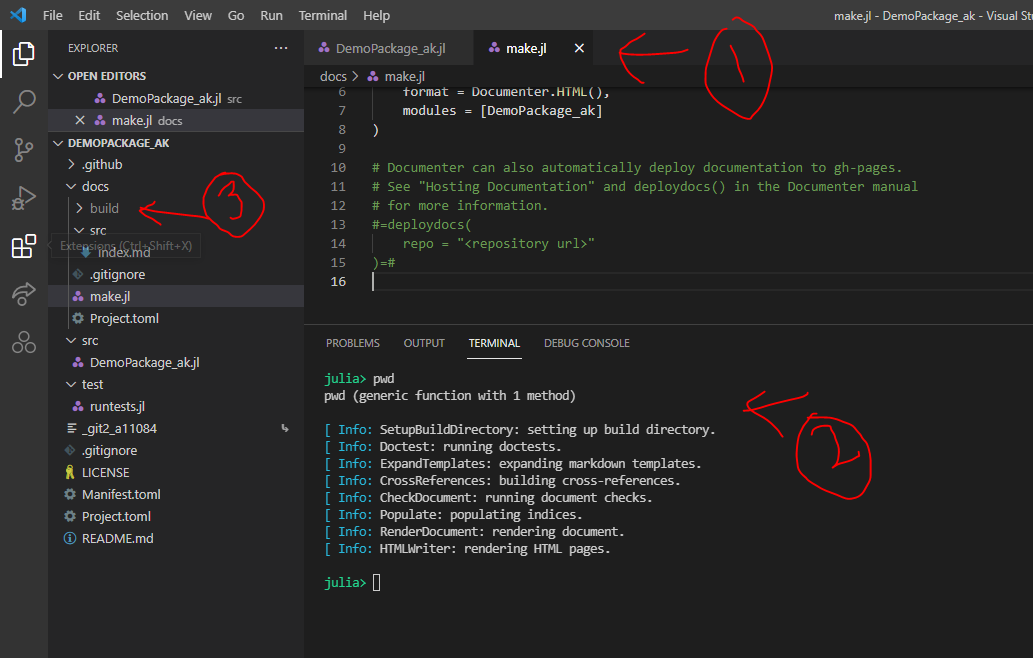
Description automatically generated with medium confidence

1. You’ll see this doc section popup

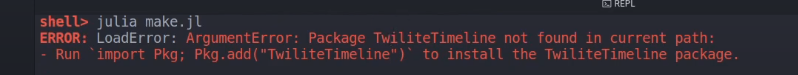
Graphical user interface, text, application

Description automatically generated

1. Now run make.jl file and you will see a new build folder would be created (2 is output, 3 is build folder)



1. If you get error like below



Add these two lines in make.jl

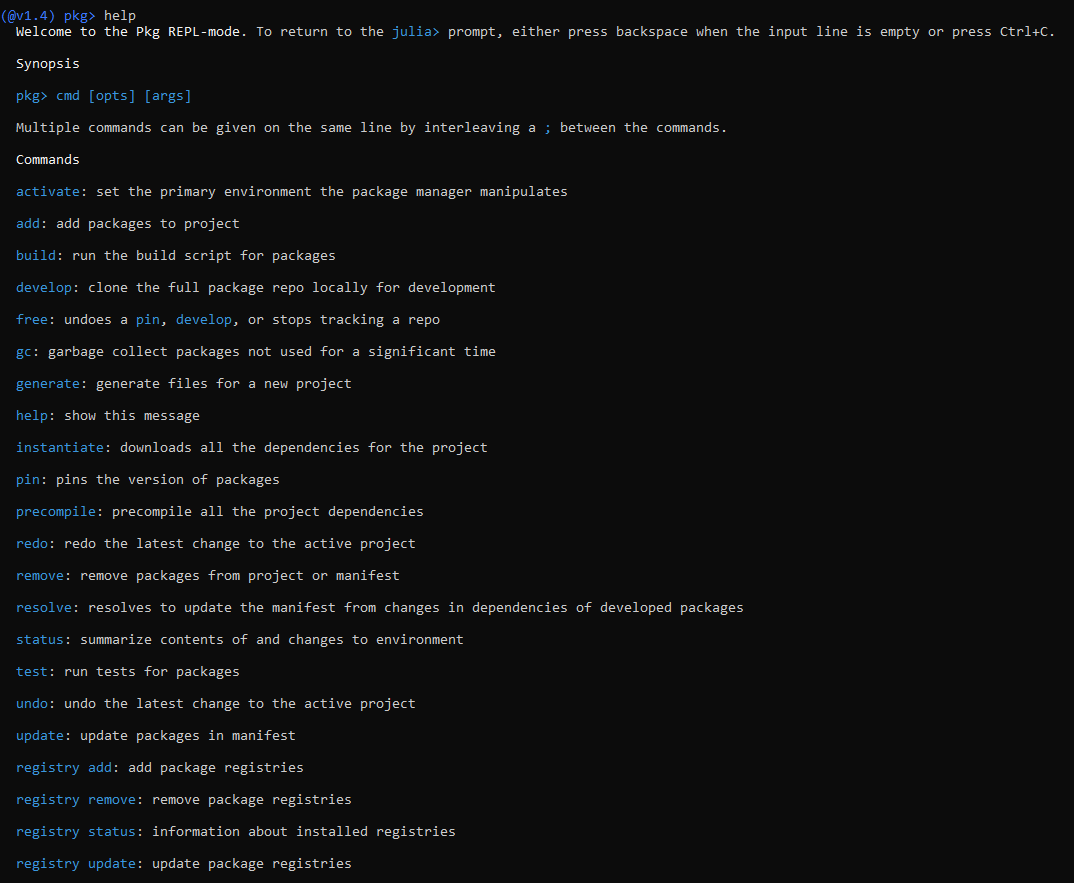
using Pkg

pkg"activate .."

# Basic Config & Troubleshooting Commands

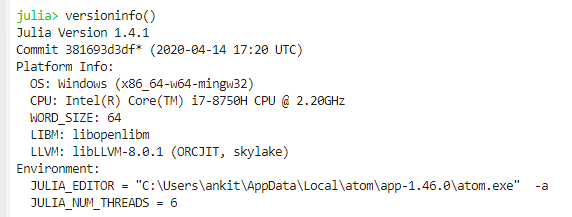
## Help

To See syntax of Package help 🡪 ] 🡪 help 🡪 enter



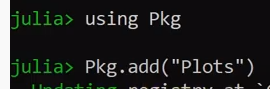
## Julia Version

versioninfo()

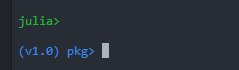


## Package management mode / PKG

Always enter – using PKG as below before entering any commands for PKG

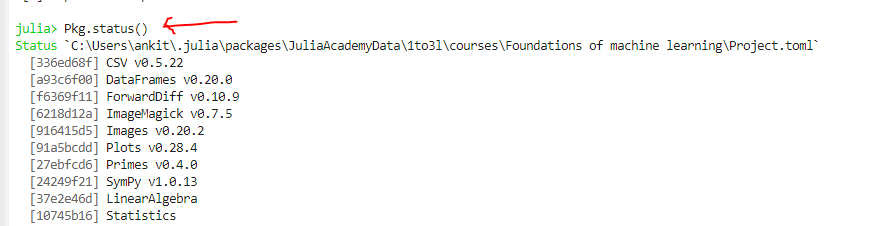


Hit ] only and it changes from Julia as below



## Pkg.status()

To check the version and all the packages installed



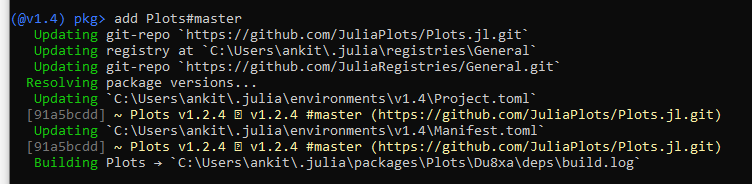
Updated a lot of above to Master



## Update Pkg – Specific Version

Add hash to define a specific version

add Plots#master



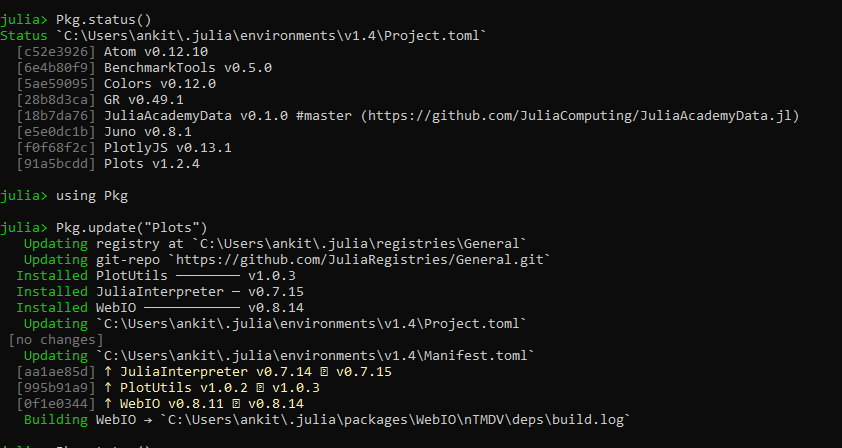
## could not load library "libGR.dll"





ENV["GRDIR"] = "C:\Users\ankit\.julia\packages\GR\cRdXQ\deps\gr"

ENV["GRDIR"] = "C:\Users\ankit\.julia\packages\GR\cRdXQ\deps\gr\bin"



# Basics of Julia

## Data Types

julia> Var1 = 43

43

julia> typeof(Var1)

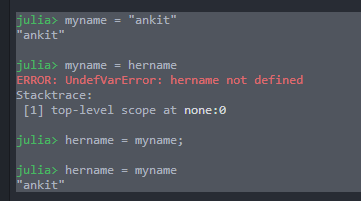
Int64

julia> pi = 3.14

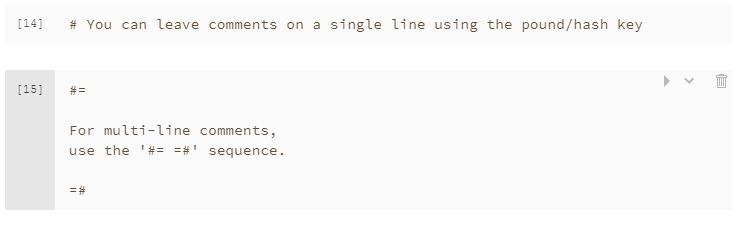
3.14

julia> typeof(pi)

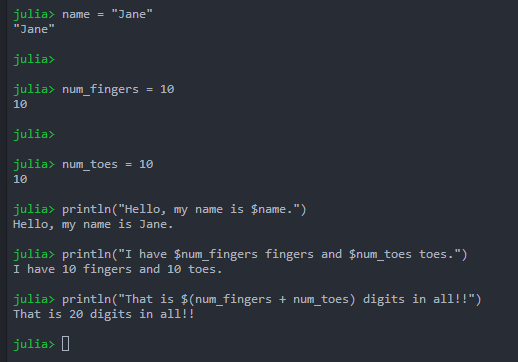
Float64



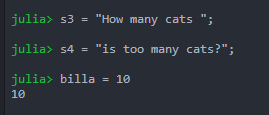
## Commenting

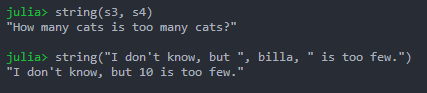


## String interpolation



## concatenate strings





Can also concatenate like below –



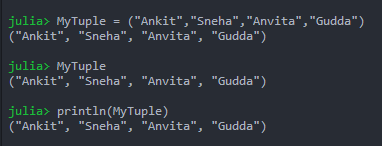
And, like this



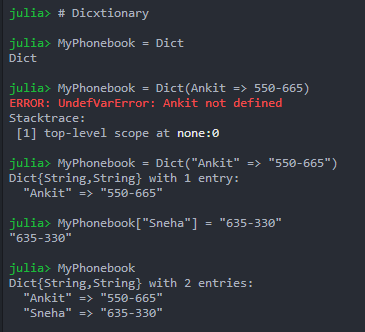
# Data Structures

## Tuples

but since tuples are immutable, we can't update it



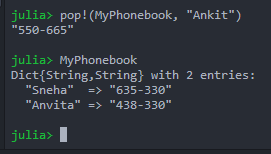
## Dictionary



each name and number is a "key" and "value" pair.

Name is key and number is value

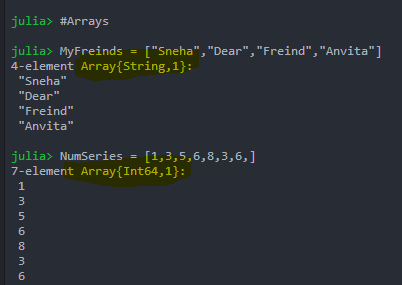
Using POP, deleted one entry from the dict



* You can use position value like in arrays to get a value in tuples, but in dictionary you have to give the key

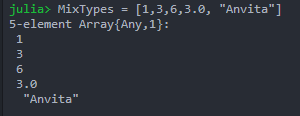
## Arrays

Arrays uses square bracket unlike tuples which uses round bracket

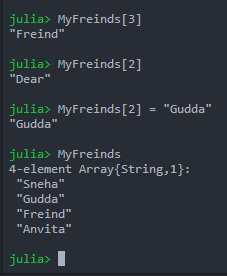


Note array automatically initialized string or int64

If we mix types then what happens



Calling a value from a position and next updating a value

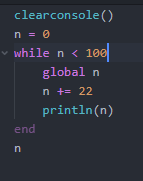
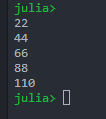


## Push & POP

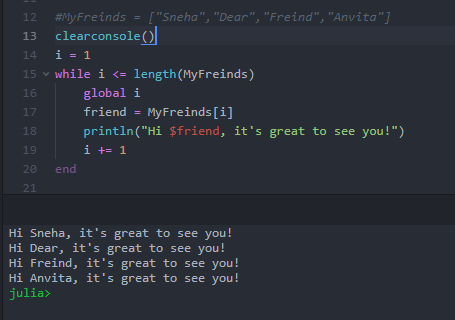
Push by defaults add in the end, and pop if you don’t specify, deleted from the end, or you can mention a location too

# Loops

## While Loop

MyFreinds = ["Sneha","Dear","Freind","Anvita"]



*#MyFreinds = ["Sneha","Dear","Freind","Anvita"]*

*clearconsole()*

*i = 1*

*while i <= length(MyFreinds)*

*global i*

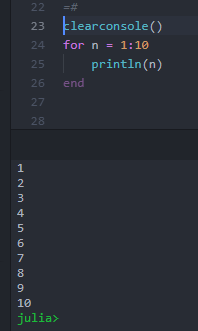
*friend = MyFreinds[i]*

*println("Hi $friend, it's great to see you!")*

*i += 1*

*end*

## For Loop



## IF ELSE

N = 25

if (N % 3 == 0) && (N % 5 == 0) # `&&` means "AND"; % computes the remainder after division

println("FizzBuzz")

elseif N % 3 == 0

println("Fizz")

elseif N % 5 == 0

println("Buzz")

else

println(N)

end

## Ternary Operator?

X=25

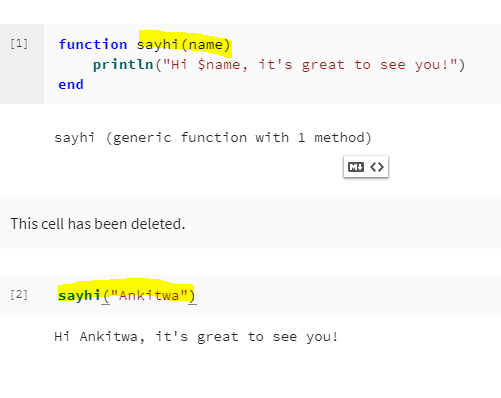
Y=26

(X < Y) ? X : Y

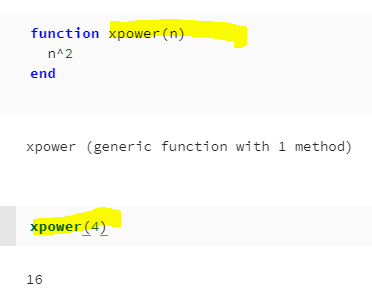
Pass x if true or y

# Functions

## 1st

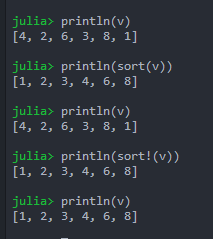


## 2nd

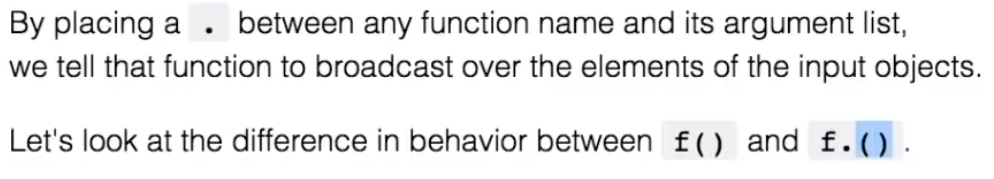


## Sort

Sort! Will alter permanenet with sort,



## Broadcasting



Difference between regular function and broadcasting function is that

Regular function treats the array as a single object but broadcasting i.e. **f.**

Will dissent each element of the array.

E.g below – each is a matrix, function of square will multiply the matrix with itself, but **F. i.e. Broadcast** will broadcast each message

MatrixA = [i + 3\*j for j in 0:2, i in 1:3]

println(MatrixA)

here is a Matrix

MatrixA = [i + 3\*j for j in 0:2, i in 1:3]

Square function

function pow2(x)

x^2

end

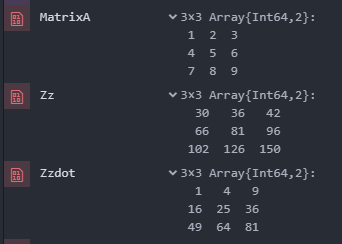
Applying pow2 on the Matrix with normal function and dot

Zz = pow2(MatrixA)

println(Zz)

Zzdot = pow2.(MatrixA)

println(Zzdot)



Notice in Zzdot, each element multiplied to itself. And in Zz it’s the whole matrix

# Packages

Until 5/10/2020

Julia has over 2000 registered packages, making packages a huge part of the Julia ecosystem.

Even so, the package ecosystem still has some growing to do. Notably, we have first class function calls to other languages, providing excellent foreign function interfaces. We can easily call into python or R, for example, with PyCall or Rcall.

This means that you don't have to wait until the Julia ecosystem is fully mature, and that moving to Julia doesn't mean you have to give up your favorite package/library from another language!

To see all available packages, check out

<https://pkg.julialang.org/> or <https://juliaobserver.com/>

## Adding Package

The first time you use a package on a given Julia installation, you need to use the package manager to explicitly add it:

Pkg.add("Example")

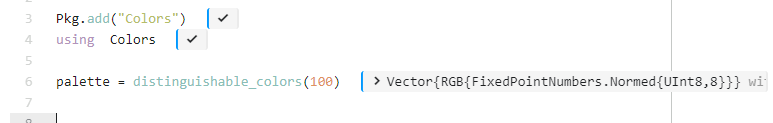
Every time you use Julia (start a new session at the REPL, or open a notebook for the first time, for example), you load the package with the using keyword

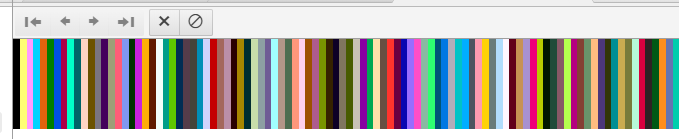
using Example

In the source code of **Example**.jl at<https://github.com/JuliaLang/Example.jl/blob/master/src/Example.jl> we see the following function declared

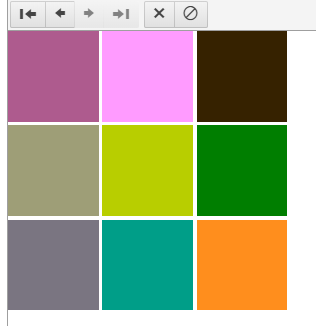
hello(who::String) = "Hello, $who"

Having loaded Example, we should now be able to call hello





rand(palette, 3,3)



# Plots

Pkg.add("Plots")

using Plots

x = -3:0.1:3 # this means -3 to 3 in increment of 0.1

f(x) = x^2

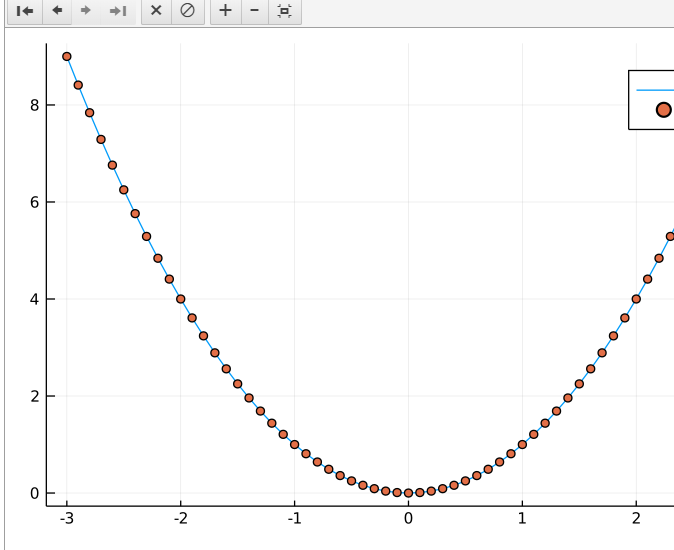
y = f.(x)

println(y)

gr() #Plots supports multiple backends — that is, libraries that actually do the drawing — all with the #same API. To start out, let's try the GR backend. You choose it with a call to gr():

plot(x,y, label="line")

scatter!(x,y, label="points") # exclamation mark on the scatter, makes it draw over the previous command line plot

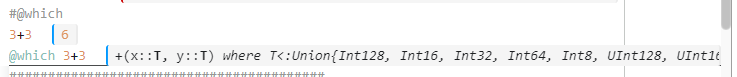


Instead of GR we will do the previous exercise now with plotlyjs

# Multiple dispatch

## @which

Tells the method

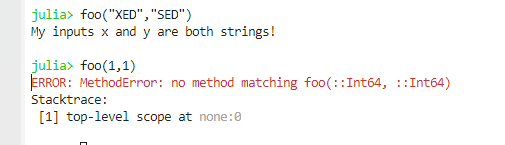


## Specifying input arguments

However, we also have the option to tell Julia explicitly what types our input arguments are allowed to have.

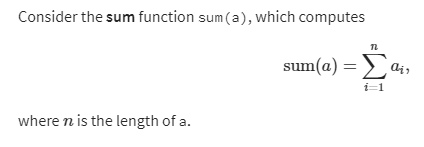
foo(x::String, y::String) = println("My inputs x and y are both strings!")

#we defines string above, so strings working fine, and integer not



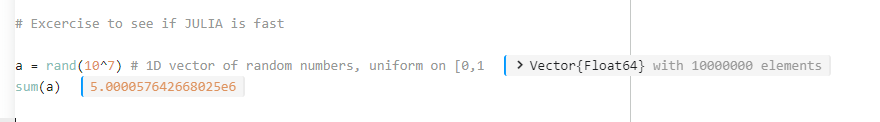
# Julia is FAST

* Define the sum function
* Implementations & benchmarking of sum in...
  + C (hand-written)
  + C (hand-written with -ffast-math)
  + python (built-in)
  + python (numpy)
  + python (hand-written)
  + Julia (built-in)
  + Julia (hand-written)
  + Julia (hand-written with SIMD)
* Summary of benchmarks



a = rand(10^7) # 1D vector of random numbers, uniform on [0,1

sum(a)

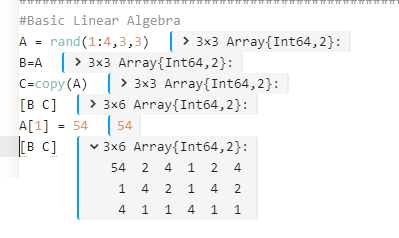


## Using Benchmark Package

# Basic Linear Algebra

Below a value was given to B (using A=B) by this way B is pointer to A and will always replicate value of A

However, when you say C = copy(A) that’s a temporary one time copy, it doesn’t reflect changes after it



Above we changed value in A, that shows in only part in [B C] but not for C

## Multiplication

# Representing Data with Models

import Pkg; Pkg.add(Pkg.PackageSpec(url="https://github.com/JuliaComputing/JuliaAcademyData.jl"))

using JuliaAcademyData;

activate("Foundations of machine learning")

#We can use the Images.jl package in Julia to load sample images from this dataset. Most of the data we will use live in the data folder in this repository.

using Images

apple = load(datapath("data/10\_100.jpg"))

banana = load(datapath("data/104\_100.jpg"))

# PROPERTIES OF THE Images

typeof(apple)

size(apple)

## Size

Size of matrix or array

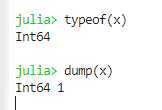


## TypeOf

Tells if a value is INT, CHAR, FLOAT etc

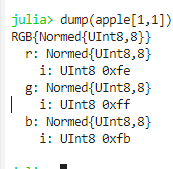


Compare with Dump

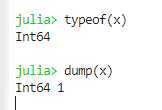


## Dump

Data type info on point or range, like for x=1, dump(x) is Int



Gives little more info that type of



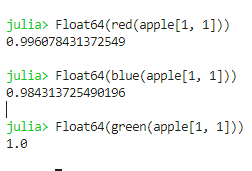
## Float64(yellow(apple[1, 1]))

Or Float64(blue(apple[1, 1]))

Or Float64(green(apple[1, 1]))

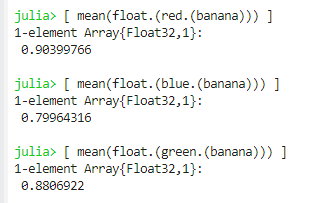
Gives a numeric value of a number at a pixel,

below three value are for white pixel:



## Mean

### Individual Mean of RGB



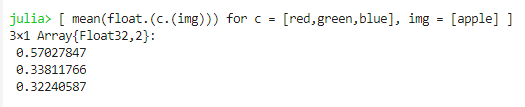
Syntax

*julia> [ mean(float.(red.(banana))) ]*

*1-element Array{Float32,1}:*

*0.90399766*

### Mean of RGB Together



*julia> [ mean(float.(c.(img))) for c = [red,green,blue], img = [apple] ]*

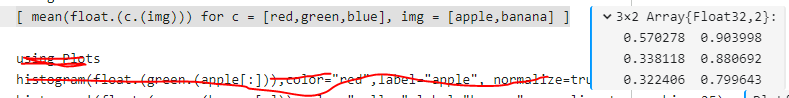
*3×1 Array{Float32,2}:*

*0.57027847*

*0.33811766*

*0.32240587*

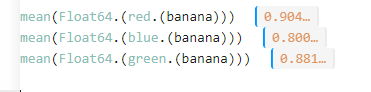
### Mean of RGB for Multiple Images

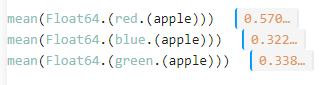


*[ mean(float.(c.(img))) for c = [red,green,blue], img = [apple,banana] ]*

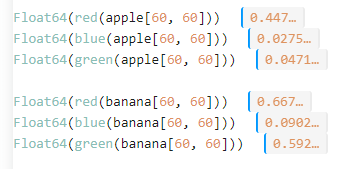
## Histogram

## Average of RGB





## Individual RGB



*Float64(red(apple[60, 60]))*

*Float64(blue(apple[60, 60]))*

*Float64(green(apple[60, 60]))*

*Float64(red(banana[60, 60]))*

*Float64(blue(banana[60, 60]))*

*Float64(green(banana[60, 60]))*

Why has Banana has more average of RED

We see that the banana's mean red value is higher than the apple's, even though the apple looks much redder. Can you guess why?

There are two reasons. One of the reasons is the background: the image of the banana has a lot more white background than the apple, and that white background has a red value of 1! In our minds we ignore the background and say, "the banana is bright yellow, the apple is dark red", but a computer just has a bundle of numbers and does not know where it should be looking.

*Syntax*

*mean(Float64.(red.(banana)))*

*mean(Float64.(blue.(banana)))*

*mean(Float64.(green.(banana)))*

# Readline()

Take user input

print("\n whats your name")

name1=readline()

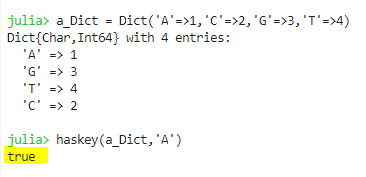
print("\n Your name is : ", name1)

# Haskey

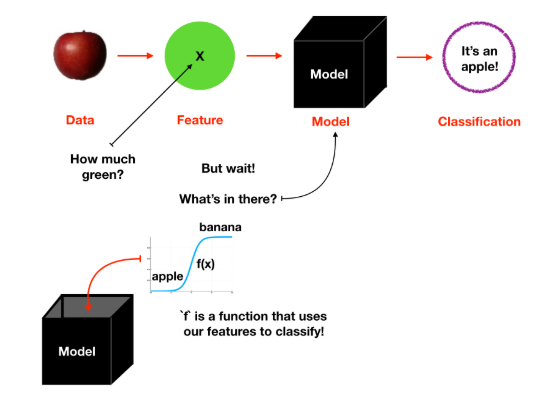
Has key tells if a value is present in a dictionary in boolean

a\_Dict = Dict('A'=>1,'C'=>2,'G'=>3,'T'=>4)

haskey(a\_Dict,'A')

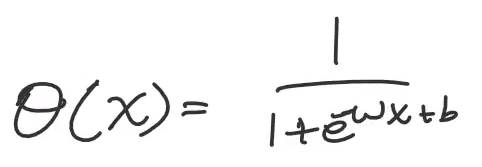


# Apple Banana



## Data Model

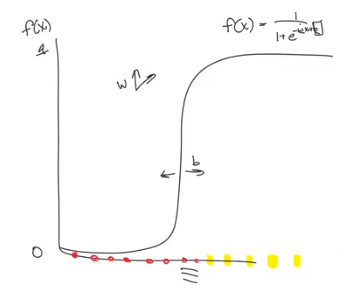
### Sigmoid Function –



**B** in the equation can be used to move the sigmoid function to justify the apple or banana



Similarly, **W** is used to make the curve less steep or steeper, that is the 0 and 1 gap.

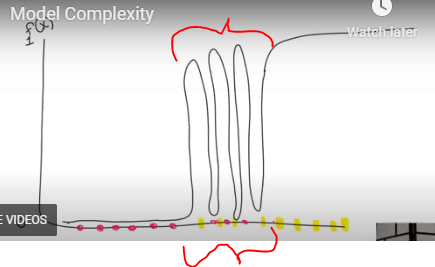


Bottomline - see above – w changes the angle of the curve and b moves the curve left or right

# Model complexity

See closely below,

* on the very left is all apple, so sigmoid is ok and is representing zero.
* On the very right is all banana, so sigmoid of 1 is good
* But in the middle it’s a mix, as shown by the middle part, so our graph will oscillate between the 1 and 0 (but this is overfitting, we don’t want that)
* To over come overfitting, we do cross validation i.e we leave the problematic points and train the model without them and see how it does.



* Decreasing the model complexity, reducing the model to fit the number is a good approach

In the last notebook, we saw that we could customize a model by adding a parameter. Doing so, we were able to fit that model to a data point. This fit was perfect, insofar as numeric would allow.

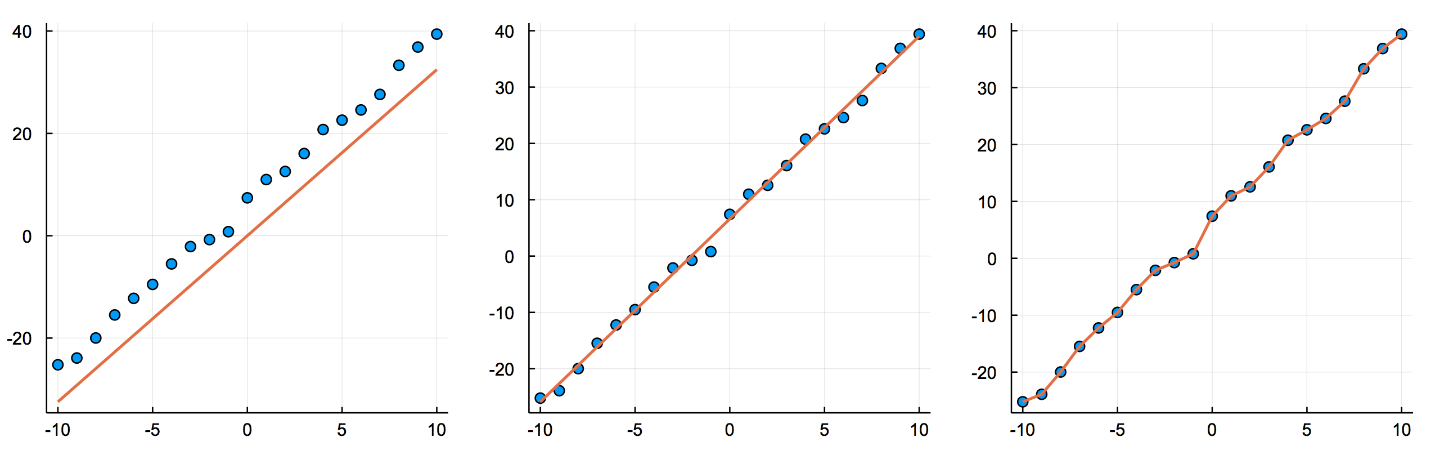
In the next notebook, we'll see that as we add more data to our data set, fitting a model to our data usually becomes more challenging and the result will be less perfect.

For one thing, we will find that we can add complexity to our model to capture added complexity in the data. We can do this by adding more parameters to our model. We'll see that for a data set with two data points, we can again get a "perfect" fit to our model by adding a second parameter to our model.

## **Overfitting**

However, we can't simply add a parameter to our model every time we add a data point to our data set, since this will lead to a phenomenon called **overfitting**.

In the image below, we depict a data set that is close to linear, and models that exhibit underfitting, fitting well, and overfitting, from left to right:



In the first image, the model accounts for the slope along which the data falls, but not the offset.

In the second image, the model accounts for both the slope and offset of the data. Adding this second parameter (the offset) to the model creates a much better fit.

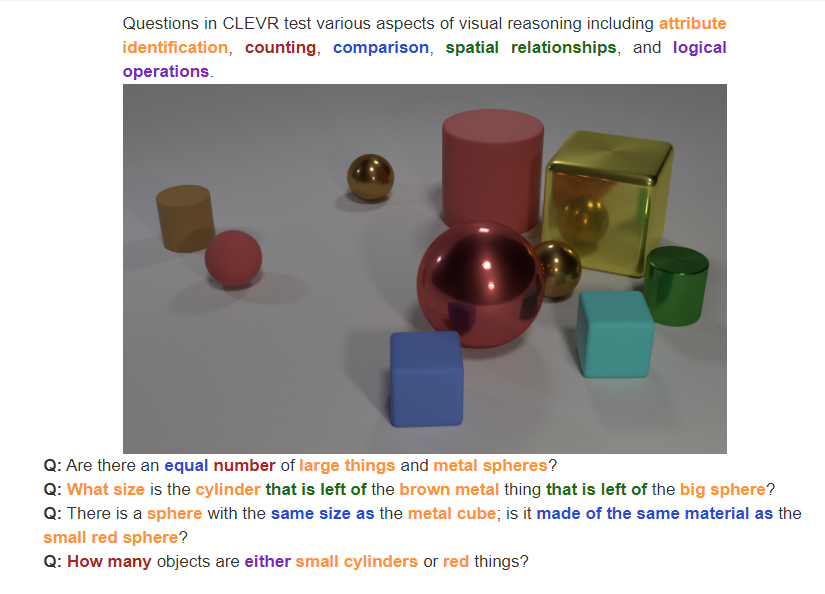
However, we can imagine that a model can have too many parameters, where we begin to fit not only the high level features of the data, but also the noise. This overfitting is depicted in the third image.

# The world of Machine Learning with Knet

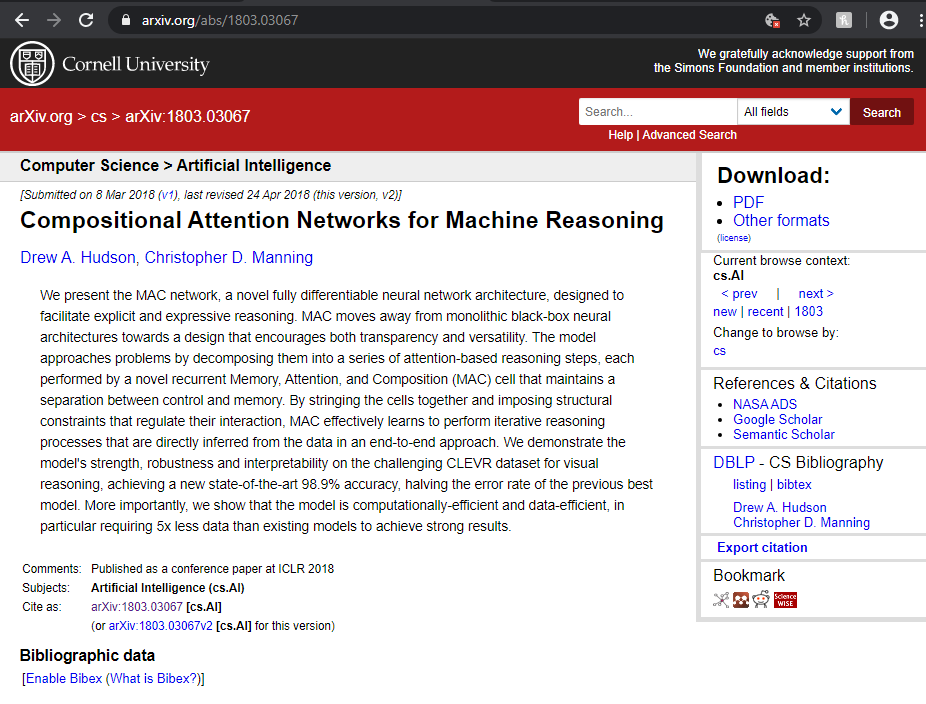
All notebooks are here on GitHub

<https://github.com/KnetML/NLPdemos>

<https://cs.stanford.edu/people/jcjohns/clevr/>



<https://arxiv.org/abs/1803.03067>



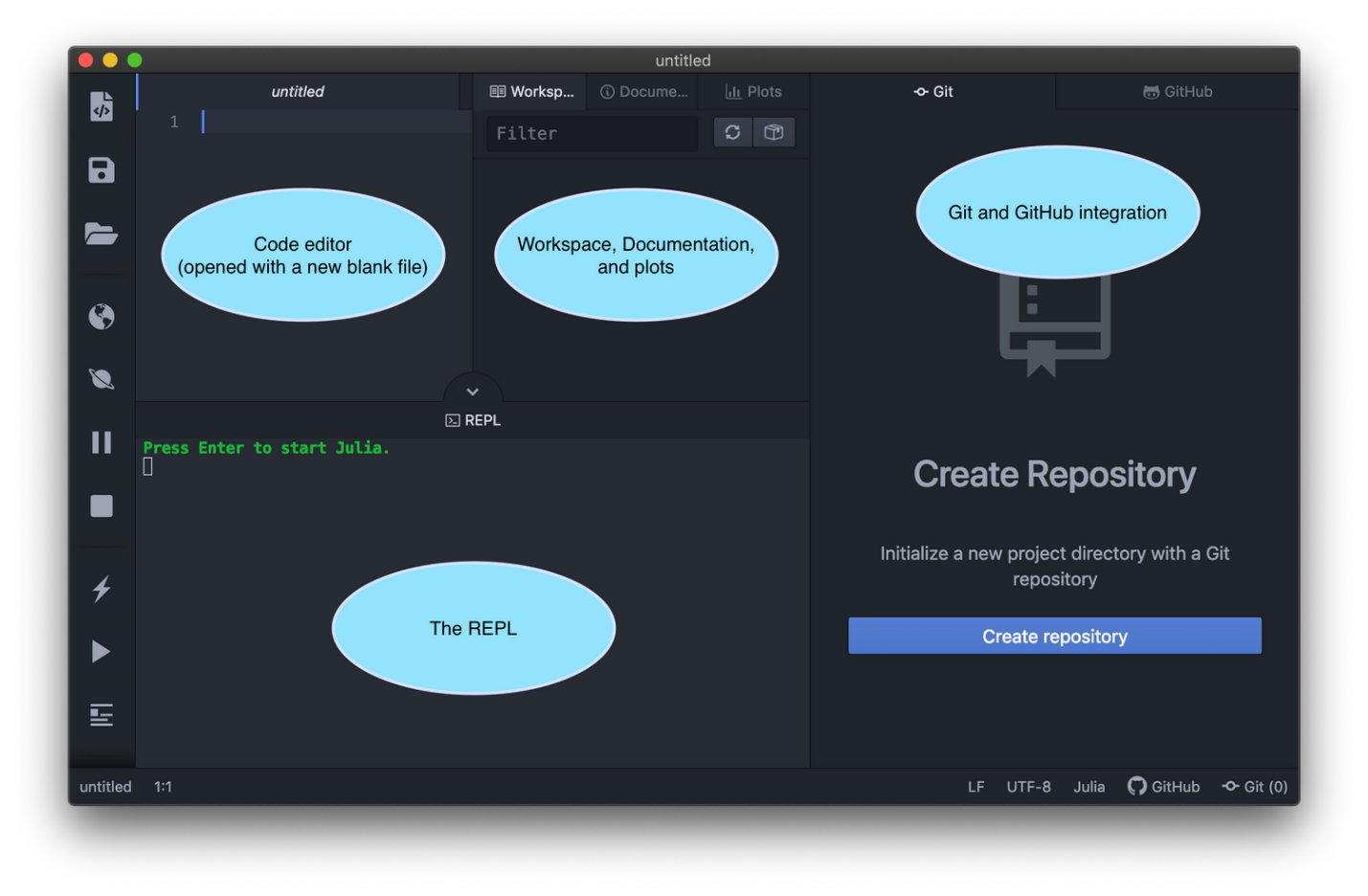
# Julia Pro - Initial 1st time

Below was a better link to setup Julia

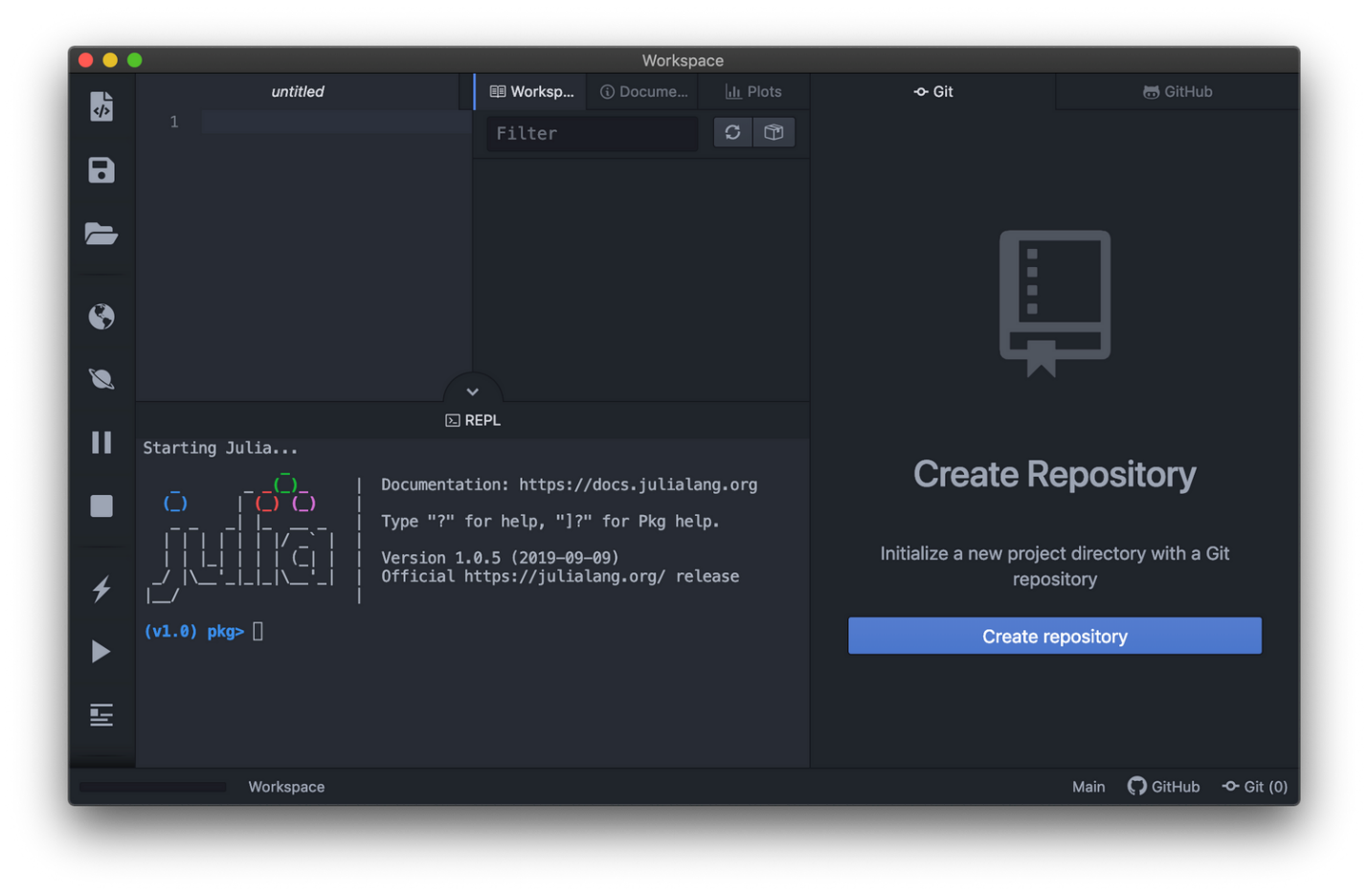
<https://www.youtube.com/watch?v=1cw-fe0n-xI>

In order for JuliaPro's package manager to work, we need to activate it. This is a three step process.

**Step 1. Launch JuliaPro.** Upon startup you'll see a four-pane layout. The most important pane for our purposes is the one on the bottom left: the REPL (stands for Read-Evaluate-Print-Loop; that is, it provides a Julia session that reads in what you write, evaluates it, prints out the result, and then loops back to read in again).

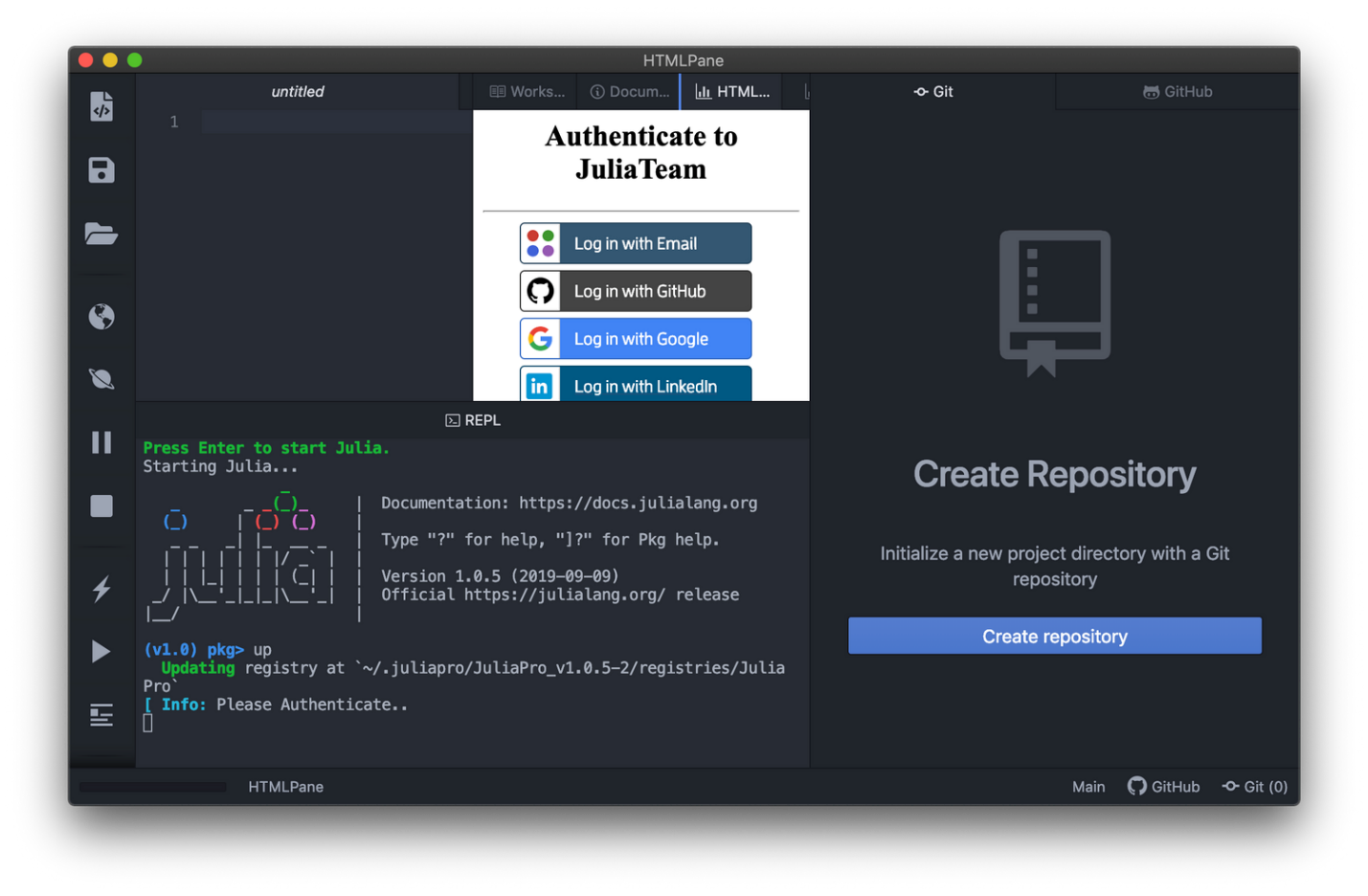


**Step 2. Start the package manager**. Click on that bottom left REPL pane and do as it says: hit enter (or return) to start Julia. You'll see a green julia> prompt pop up where you can start executing Julia code. You can of course also start using JuliaPro as a full-fledged IDE, but all we want to do here, though, is to initiate a connection with the package manager. To do that, type a ] at the prompt. This will "change modes" into the package management mode:

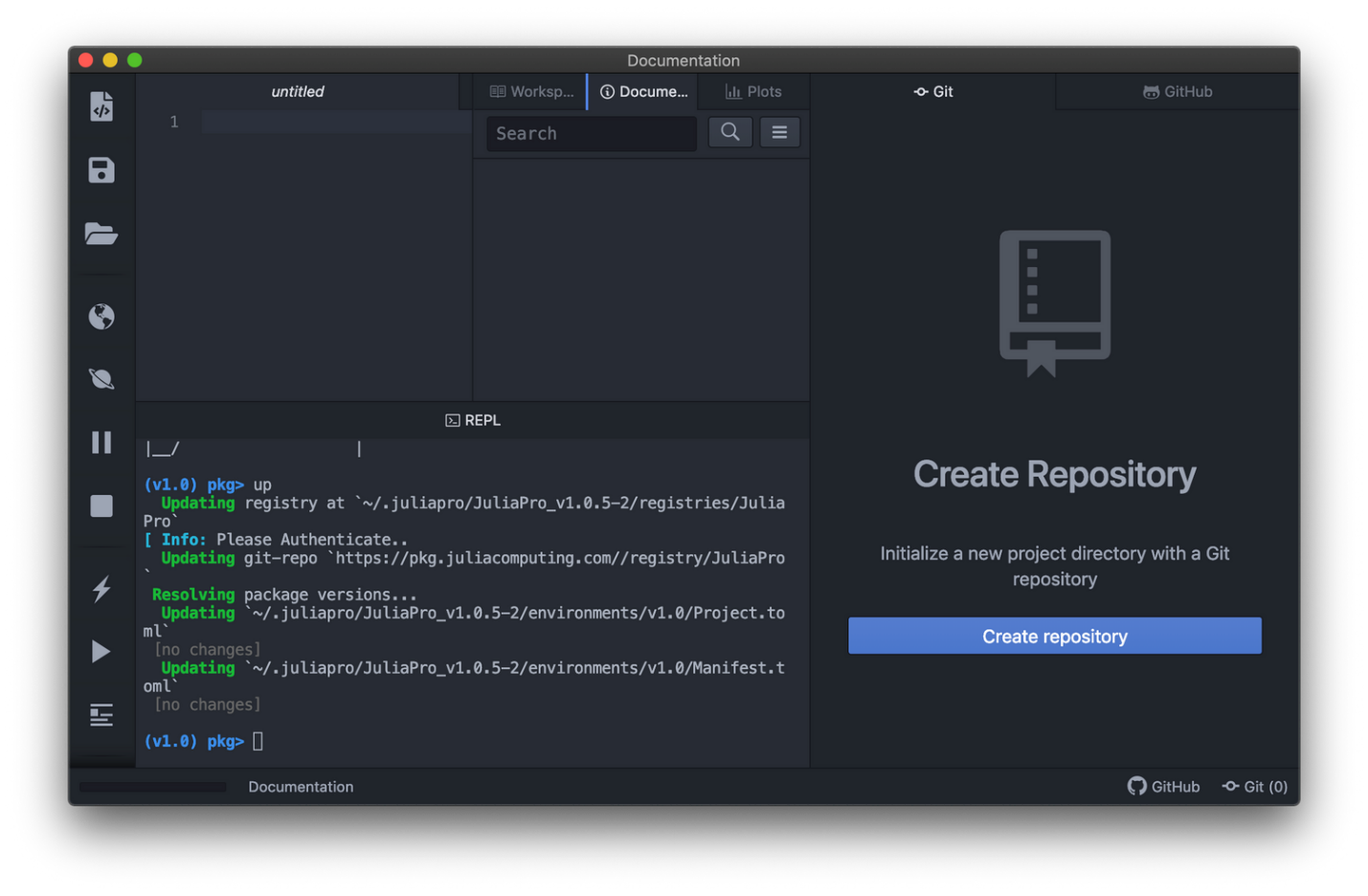


**Step 3. Authenticate.** Run `update` to connect to the package manager. Type the word up at the (1.0) pkg> prompt and up will pop an authentication window. Log in however you prefer, either with an email and password or with any one of the providers as is convenient — you can use the same login method you used when you first downloaded JuliaPro (but you can use a different one, too).

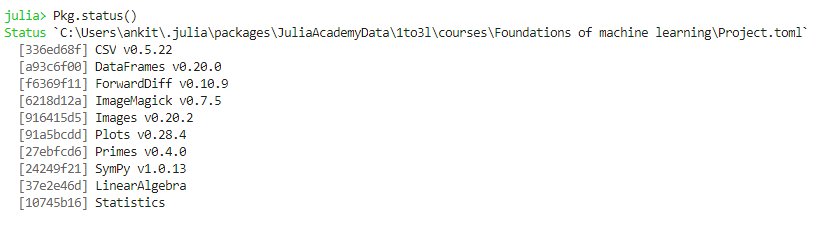
**Note: It is strongly recommended that you use GitHub to Log in as some of the other methods may have trouble authenticating based on your operating system.**



**Step 4. You're done!** Successfully authenticating might look like this: it'll connect to the package registry and say there are no new changes. It may also actually update — but it's typically already installed with the latest versions of all the packages.



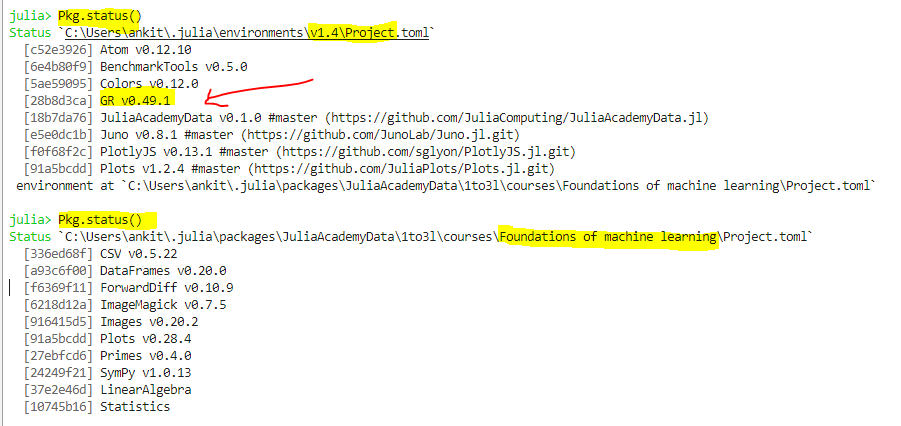




# Foundations of machine learning\Project.toml`

<https://github.com/JuliaComputing/JuliaAcademyData.jl/issues/4>

<https://github.com/JuliaPlots/Plots.jl/issues/993>



using Plots;

┌ Warning: Error requiring Juno from PlotThemes:

│ LoadError: MethodError: no method matching hex(::UInt32, ::Int64, ::Bool)

│ You may have intended to import Base.hex

│ Stacktrace:

│ [1] (::PlotThemes.var"#22#28")(::Pair{String,UInt32}) at .\none:0

│ [2] iterate at .\generator.jl:47 [inlined]

│ [3] \_all at .\reduce.jl:819 [inlined]

│ [4] all at .\reduce.jl:815 [inlined]

│ [5] Dict(::Base.Generator{Dict{String,UInt32},PlotThemes.var"#22#28"}) at .\dict.jl:130

│ [6] top-level scope at C:\Users\XXXX\.julia\packages\PlotThemes\VNU1I\src\juno\_smart.jl:5

│ [7] include(::Module, ::String) at .\Base.jl:377

│ [8] include(::String) at C:\Users\ankit\.julia\packages\PlotThemes\VNU1I\src\PlotThemes.jl:3

│ [9] top-level scope at C:\Users\ankit\OneDrive\Julia\JuliaWork\AppleBanana.jl:36

│ [10] eval at .\boot.jl:331 [inlined]

│ [11] eval at C:\Users\ankit\.julia\packages\PlotThemes\VNU1I\src\PlotThemes.jl:3 [inlined]

│ [12] (::PlotThemes.var"#18#21")() at C:\Users\ankit\.julia\packages\Requires\qy6zC\src\require.jl:85

│ [13] err(::Any, ::Module, ::String) at C:\Users\ankit\.julia\packages\Requires\qy6zC\src\require.jl:42

│ [14] (::PlotThemes.var"#17#20")() at C:\Users\ankit\.julia\packages\Requires\qy6zC\src\require.jl:84

│ [15] withpath(::Any, ::String) at C:\Users\ankit\.julia\packages\Requires\qy6zC\src\require.jl:32

│ [16] (::PlotThemes.var"#16#19")() at C:\Users\ankit\.julia\packages\Requires\qy6zC\src\require.jl:83

│ [17] listenpkg(::Any, ::Base.PkgId) at C:\Users\ankit\.julia\packages\Requires\qy6zC\src\require.jl:15

│ [18] macro expansion at C:\Users\ankit\.julia\packages\Requires\qy6zC\src\require.jl:81 [inlined]

│ [19] \_\_init\_\_() at C:\Users\ankit\.julia\packages\PlotThemes\VNU1I\src\PlotThemes.jl:63

│ [20] \_include\_from\_serialized(::String, ::Array{Any,1}) at .\loading.jl:697

│ [21] \_require\_search\_from\_serialized(::Base.PkgId, ::String) at .\loading.jl:781

│ [22] \_tryrequire\_from\_serialized(::Base.PkgId, ::UInt64, ::String) at .\loading.jl:712

│ [23] \_require\_search\_from\_serialized(::Base.PkgId, ::String) at .\loading.jl:770

│ [24] \_require(::Base.PkgId) at .\loading.jl:1006

│ [25] require(::Base.PkgId) at .\loading.jl:927

│ [26] require(::Module, ::Symbol) at .\loading.jl:922

│ [27] include\_string(::Module, ::String, ::String) at .\loading.jl:1080

│ [28] include\_string(::Module, ::String, ::String, ::Int64) at C:\Users\ankit\.julia\packages\CodeTools\kosGY\src\eval.jl:30

│ [29] (::Atom.var"#188#192"{String,Int64,String,Bool})() at C:\Users\ankit\.julia\packages\Atom\wlPiw\src\eval.jl:111

│ [30] withpath(::Atom.var"#188#192"{String,Int64,String,Bool}, ::String) at C:\Users\ankit\.julia\packages\CodeTools\kosGY\src\utils.jl:30

│ [31] withpath(::Function, ::String) at C:\Users\ankit\.julia\packages\Atom\wlPiw\src\eval.jl:9

│ [32] #187 at C:\Users\ankit\.julia\packages\Atom\wlPiw\src\eval.jl:110 [inlined]

│ [33] with\_logstate(::Atom.var"#187#191"{String,Int64,String,Bool}, ::Base.CoreLogging.LogState) at .\logging.jl:398

│ [34] with\_logger at .\logging.jl:505 [inlined]

│ [35] #186 at C:\Users\ankit\.julia\packages\Atom\wlPiw\src\eval.jl:109 [inlined]

│ [36] hideprompt(::Atom.var"#186#190"{String,Int64,String,Bool}) at C:\Users\ankit\.julia\packages\Atom\wlPiw\src\repl.jl:140

│ [37] macro expansion at C:\Users\ankit\.julia\packages\Atom\wlPiw\src\eval.jl:108 [inlined]

│ [38] macro expansion at C:\Users\ankit\.julia\packages\Media\ItEPc\src\dynamic.jl:24 [inlined]

│ [39] eval(::String, ::Int64, ::String, ::String, ::Bool) at C:\Users\ankit\.julia\packages\Atom\wlPiw\src\eval.jl:105

│ [40] macro expansion at C:\Users\ankit\.julia\packages\Atom\wlPiw\src\eval.jl:39 [inlined]

│ [41] (::Atom.var"#172#173")() at .\task.jl:358

│ in expression starting at C:\Users\ankit\.julia\packages\PlotThemes\VNU1I\src\juno\_smart.jl:3

└ @ Requires C:\Users\ankit\.julia\packages\Requires\qy6zC\src\require.jl:44

plot(x -> σ(x, w, b), xlim=(-0,1), ylim=(-0.1,1.1), label="model", legend=:topleft, lw=3)

could not load library "libGR.dll"

The specified module could not be found.

setcharheight at GR.jl:1263 [inlined]

gr\_set\_font(::Plots.Font; halign::Symbol, valign::Symbol, color::RGB{Normed{UInt8,8}}, rotation::Int64) at gr.jl:386

(::Plots.var"#gr\_set\_font##kw")(::NamedTuple{(:halign, :valign, :rotation),Tuple{Symbol,Symbol,Int64}}, ::typeof(Plots.gr\_set\_font), ::Plots.Font) at gr.jl:385

gr\_set\_xticks\_font(::Plots.Subplot{Plots.GRBackend}) at gr.jl:723

\_update\_min\_padding!(::Plots.Subplot{Plots.GRBackend}) at gr.jl:904

iterate at generator.jl:47 [inlined]

\_collect(::Array{AbstractLayout,2}, ::Base.Generator{Array{AbstractLayout,2},typeof(Plots.\_update\_min\_padding!)}, ::Base.EltypeUnknown, ::Base.HasShape{2}) at array.jl:678

collect\_similar(::Array{AbstractLayout,2}, ::Base.Generator{Array{AbstractLayout,2},typeof(Plots.\_update\_min\_padding!)}) at array.jl:607

map(::Function, ::Array{AbstractLayout,2}) at abstractarray.jl:2072

\_update\_min\_padding!(::Plots.GridLayout) at layouts.jl:310

prepare\_output(::Plots.Plot{Plots.GRBackend}) at plot.jl:261

showjuno(::IOContext{Base.GenericIOBuffer{Array{UInt8,1}}}, ::MIME{Symbol("text/html")}, ::Plots.Plot{Plots.GRBackend}) at output.jl:247

show(::IOContext{Base.GenericIOBuffer{Array{UInt8,1}}}, ::MIME{Symbol("application/prs.juno.plotpane+html")}, ::Plots.Plot{Plots.GRBackend}) at output.jl:205

show(::IOContext{Base.GenericIOBuffer{Array{UInt8,1}}}, ::String, ::Plots.Plot{Plots.GRBackend}) at multimedia.jl:109

displayinplotpane(::Plots.Plot{Plots.GRBackend}) at showdisplay.jl:51

displayandrender(::Plots.Plot{Plots.GRBackend}) at showdisplay.jl:131

(::Atom.var"#189#193"{String})() at eval.jl:126

#invokelatest#1 at essentials.jl:712 [inlined]

invokelatest at essentials.jl:711 [inlined]

macro expansion at dynamic.jl:24 [inlined]

eval(::String, ::Int64, ::String, ::String, ::Bool) at eval.jl:105

macro expansion at eval.jl:39 [inlined]

(::Atom.var"#172#173")() at task.jl:358

* add Plots#master
* add GR

ERROR: SystemError: opening file "C:\\Users\\XXXX\\.julia\\packages\\JuliaAcademyData\\1to3l\\courses\\Foundations of machine learning\\Project.toml": Permission denied