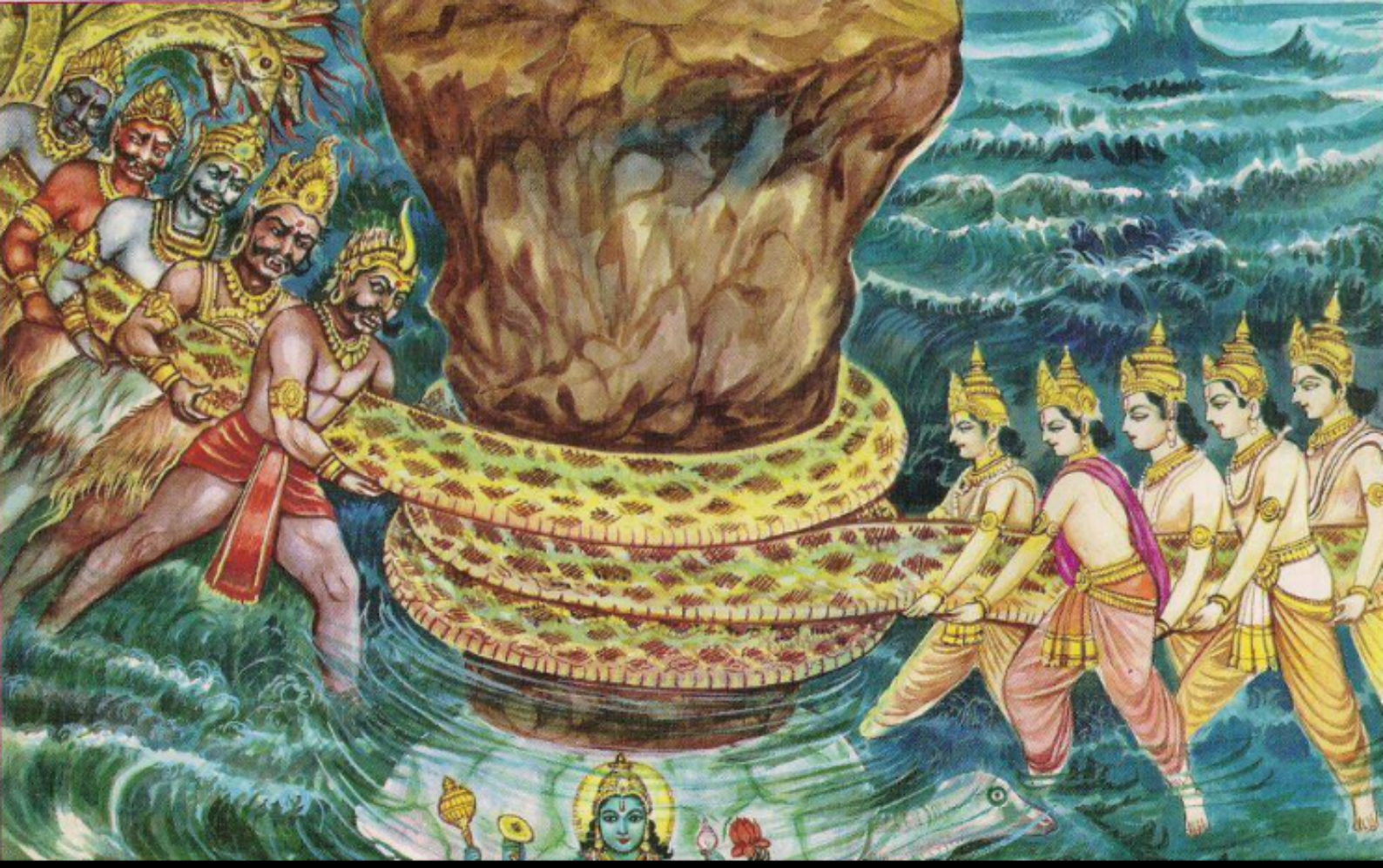


Introduction to DataScience

Learn Julia Programming, Math & Data Science from Scratch.

Karthikeyan A K

Introduction to Datascience

Learn Julia Programming, Math & Datascience from Scratch.

Karthikeyan A K

Table of Contents

Preface

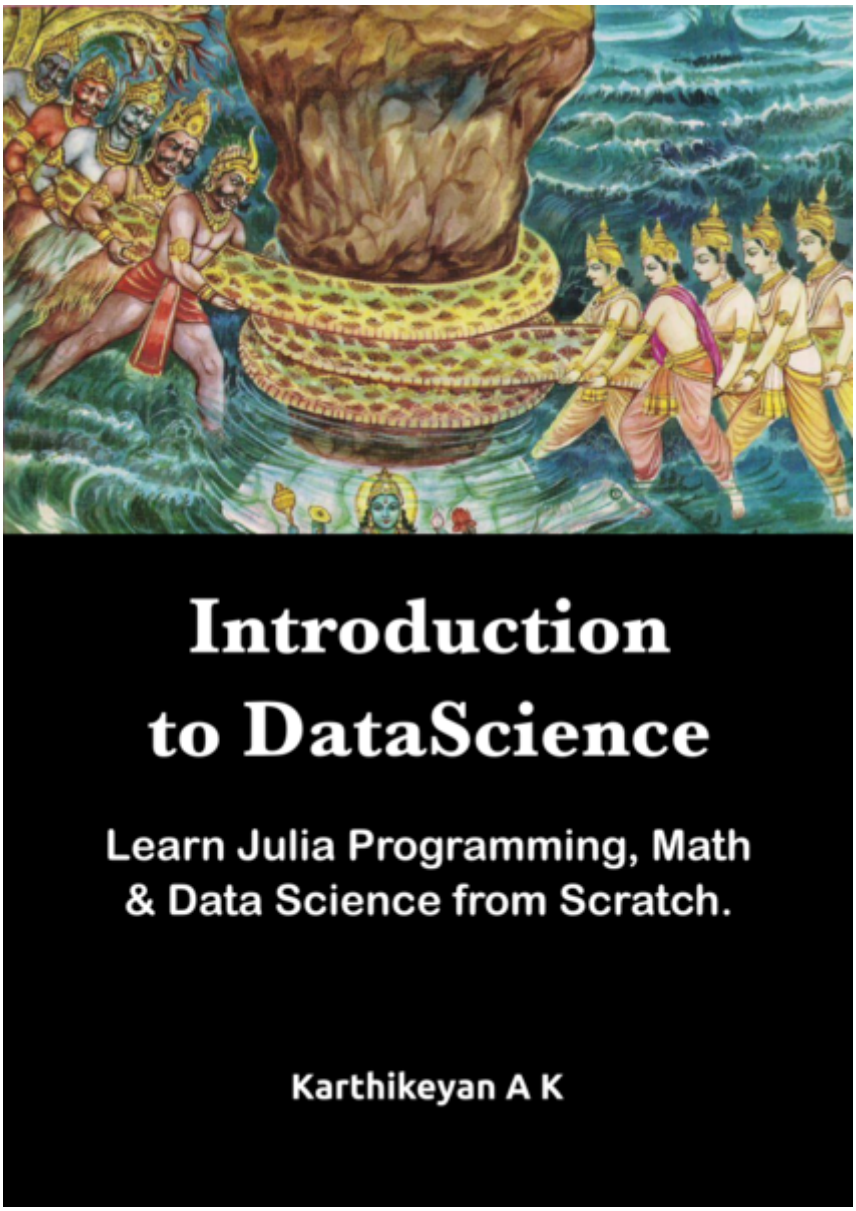
I was emboldened to write this book after my video series called Data Science With Julia[1] got some traction. That too after a tweet about Decision Tree[2] was liked by Julia Language itself. So I thought why not give it more?

This book should be seen as my attempt to explain Data Science to my self and nothing more. Will this book rise to professional stature is yet to be seen.

[1] https://www.youtube.com/playlist?list=PLe1T0uBrDrfOLQlomF_4AxHa4LX0wsCXa

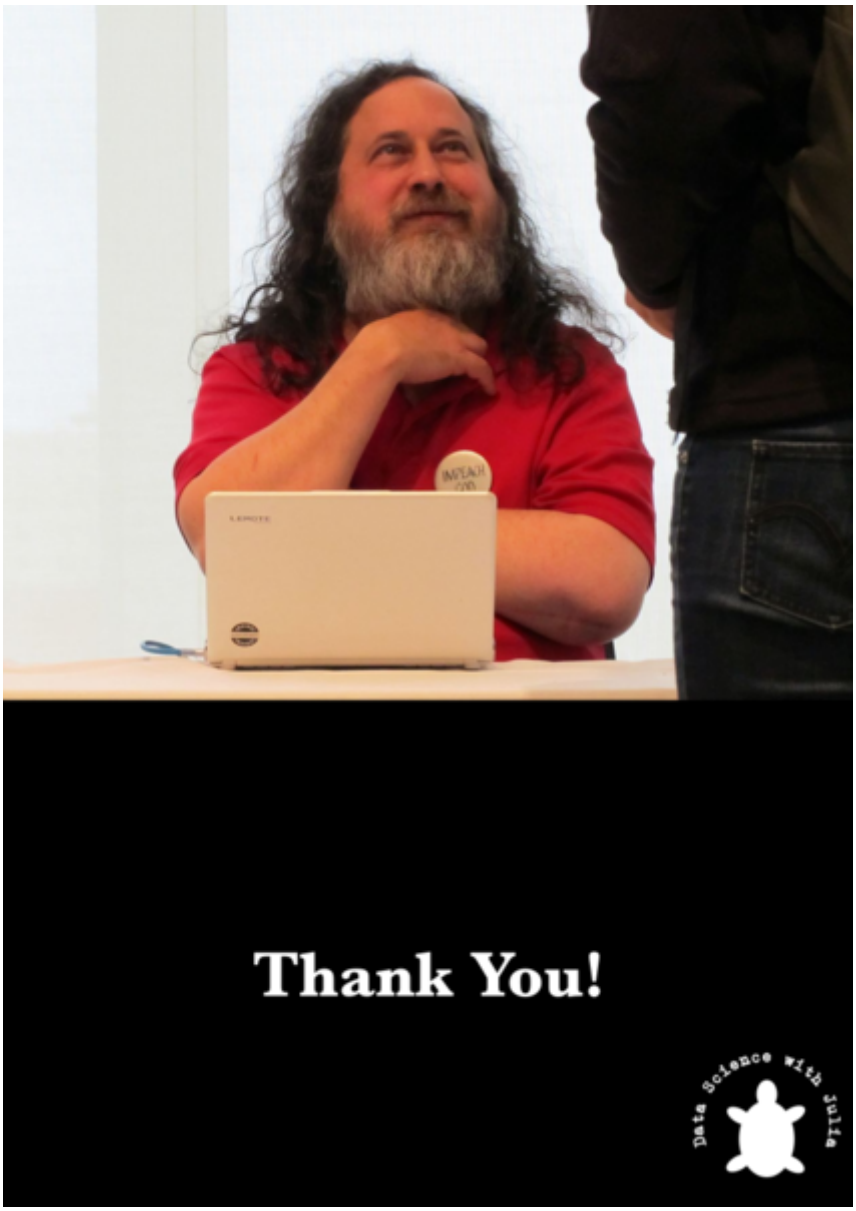
[2] https://twitter.com/karthik_ak/status/1429767974064324608

Front Cover



The front cover showcases a scene from Indian mythology where the forces of Good (Devar's) and that of Evil (Asuran's) churn the cosmic ocean revealing things like poison and elixir. In a similar way a Data Scientist can churn the vast amount of data that he has at his disposal for good purpose like finding out a new drug that might work, or for evil purpose like tracking and invading privacy of some one.

Back Cover



We thank Richard M. Stallman (https://stallman.org) and Free Software Foundation (https://fsf.org) for making this book possible.

Chapter 1. What you need to know

1.1. GNU/Linux

You need to know GNU/Linux if you have not used it, one of the best places to learn it is https://linuxjourney.com.

1.2. Math

Data Science is the a place where data processing meets computer science. Computers are good and are very fast at math, and data science is math. To know math, one can look into the courses offered by Khan Academy[3]. One can go through these courses

- Precalculus
- Calculus
- Matrix
- Probability and Statistics

One also read this book Mathematics for Machine Learning [mml].

[3] https://khanacademy.org

Chapter 2. What you need to have?

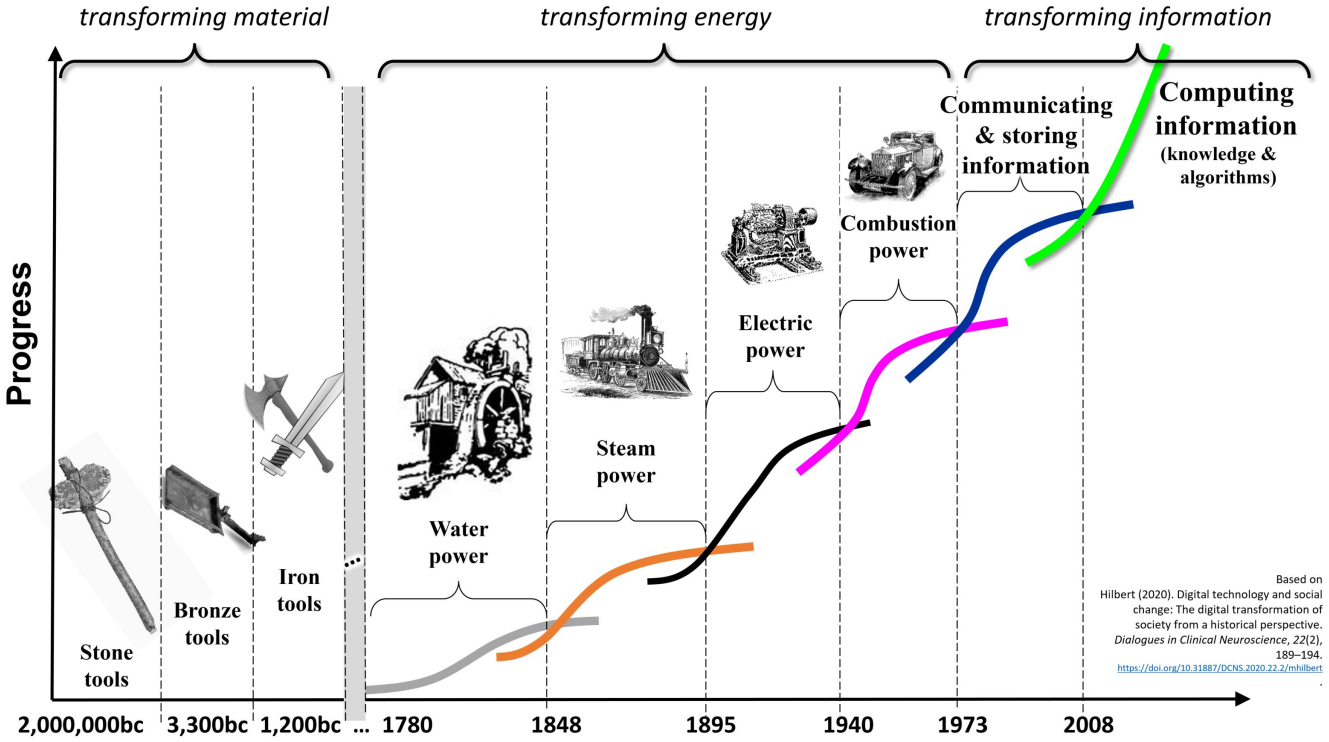
A good decent powered computer might be needed to run programs in this book. I would say its preferable to have a GNU/Linux[4] machines so that you can explore the field of data science.

[4] https://www.getgnulinux.org/

Datascience

Chapter 3. What is Datascience?

There is lot of Data, in fact we have coined the term data explosion, and more times than we realize these data indicate something valuable. With data people have become great stock traders[5], they have made machines win competitions that were thought only humans could win it[6].



It turns out that if we can use math on lot's of data with clever computer science, we could do things that were once thought impossible, and there is lot's of data out there.

Data Science is a field where we use computer science to make sense of huge amounts data around us. We try to make it humanly understandable, we try to predict things even though we may not understand why somethings happen. That's data science.

This might be the cusp of super intelligence, surely this is a new age of data. So welcome to this wonderful universe!

[5] https://www.amazon.com/Man-Who-Solved-Market-Revolution/dp/073521798X

[6] https://www.techrepublic.com/article/ibm-watson-the-inside-story-of-how-the-jeopardy-winning-supercomputer-was-born-and-what-it-wants-to-do-next/

Chapter 4. Stages in Data Science

In professional environments Data Science may not be simple. It requires lot of stages, when I say stages, don't imagine it as a waterfall model where each of them is a compartment, imagine them as a fuzzy thing where one overlaps another. Enough of work in one may lead to another. This section describes those stages.

4.1. Gathering Data

One of the first stage is gathering data. In some projects there could be data and in some one might not have data. You must devise a way to gather data. In some cases the client will present you with data but it will not be the right one.

Even if you gather the right data, it might be in a very messy state, not suitable for any purpose.

4.2. Data Wrangling

Data wrangling is a phase where you try to modify the data so that it suits your purpose. May be you need to collect tweets and label it as positive or negative sentiment. May be you use data from a spreadsheet where lot's of things are missing, what to do about it?

You need to understand the task at hand and communicate with people and come up with plan to make the data more suitable for applying math on it.

4.3. Statistics

Statistics will be applied on data during this phase. You might count number of words that occur in a particular text to understand something about it. You might summarize some numerical values into mean, median, mode, inter quartile range (IQR) etc, and try to understand make sense about it and so on. Usually statistics if performed on wrangled and cleaned up data.

4.4. Visualization

In real world Data Science most of the problem's are solved just by visualization. Humans are visual creatures and they can get it easily when something is presented in visual format. I would even argue visualization is the most important thing in Data Science.

4.5. Machine Learning (ML)

When you do Statistics and Visualization, most problem yield and become understandable. If not we must go in for machine learning, where machines and algorithms are set to task to learn about data to predict something about it, or to categorize the data and so on.

4.5.1. Feature Engineering

One of the main things in machine learning is feature engineering. Think of the ship Titanic, when

it drowned some survived and others died. Do you think the survival rate of a person depended on his name? The color of dress he wore? Or do you think it depended on his age, sex, what class he travelled? These are the things you need to think in feature engineering.

Ideally if the right feature are picked and given to a machine learning system, it would train fast and give right result.

4.5.2. Machine Learning

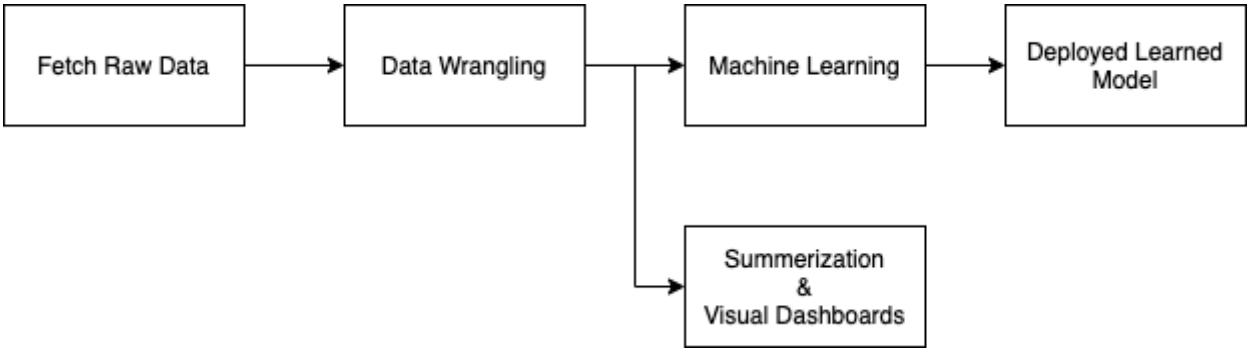
Machine learning is the phase where you give (highly modified) data to a learning system hoping that it will learn about it and produce a good result. You may be need to try out many ML algorithms before settling on the optimal one.

4.5.3. Neural Networks

When the data available to you is really large, and if you are unable to pick up the features, then one can go into Neural Networks. Neural networks are loosely modeled after neurons[7] in our brain, and can train themselves without much human intervention.

4.6. Automation

When you have arrived at an agreeable solution following the techniques presented in the sections above, you need to automate the process like building a good programming library and hosting it on computers to automate the task. An automated task could do the things that are shown below:



This stage by stage approach of transforming data is called pipelining.

4.7. Scaling

Usually in projects that needs pipelining, and that deal with lot of data volume, it also needs scaling. That is what would you do if there is flood of data into your sleepy servers? If you have got lot of servers running your algorithms to process your data, what will happen to those servers when the data dries up? You don't want to be paying high bills for unused servers. These are all the things you will worry in the scaling phase.

But this book is not about scaling. But it could cover in the future.

[7] https://en.wikipedia.org/wiki/Neuron

Chapter 5. Predictive And Descriptive Analysis

There are two ways to analyze data. The first one is very highly romanticized prediction. That is given a set of inputs you are able to predict what the outcome of the inputs would be by making machines learn from the sampled data. This field is called Machine Learning, artificial Intelligence and so on.

The other more essential field is called descriptive analysis. That is some event has happened, then you pour over the data in and out why things have happened that way and what can you learn from it.

In practical sense descriptive analysis is highly important for businesses as they would like to know why things happened in such a way in times of good and bad. A complete understanding of any subject is desired if one wants to be alpha in it and be able to better adjust to changing events and come out at the top.

5.1. Descriptive & Predictive Analysis - London Cholera

one may find the wikipedia article here https://en.wikipedia.org/wiki/1854_Broad_Street_cholera_outbreak

London once had a cholera outbreak and none seemed to know how it happened, at that time in mid 1850's, none seemed to know how cholera occurred, some even thought that one contracted cholera if they breathe in foul air and it could be avoided if we could breathe in scents.

Figure 1. John Snow

A person named John snow did scientific analysis of the outbreak, he plotted the number of deaths of cholera onto map of London as shown below:

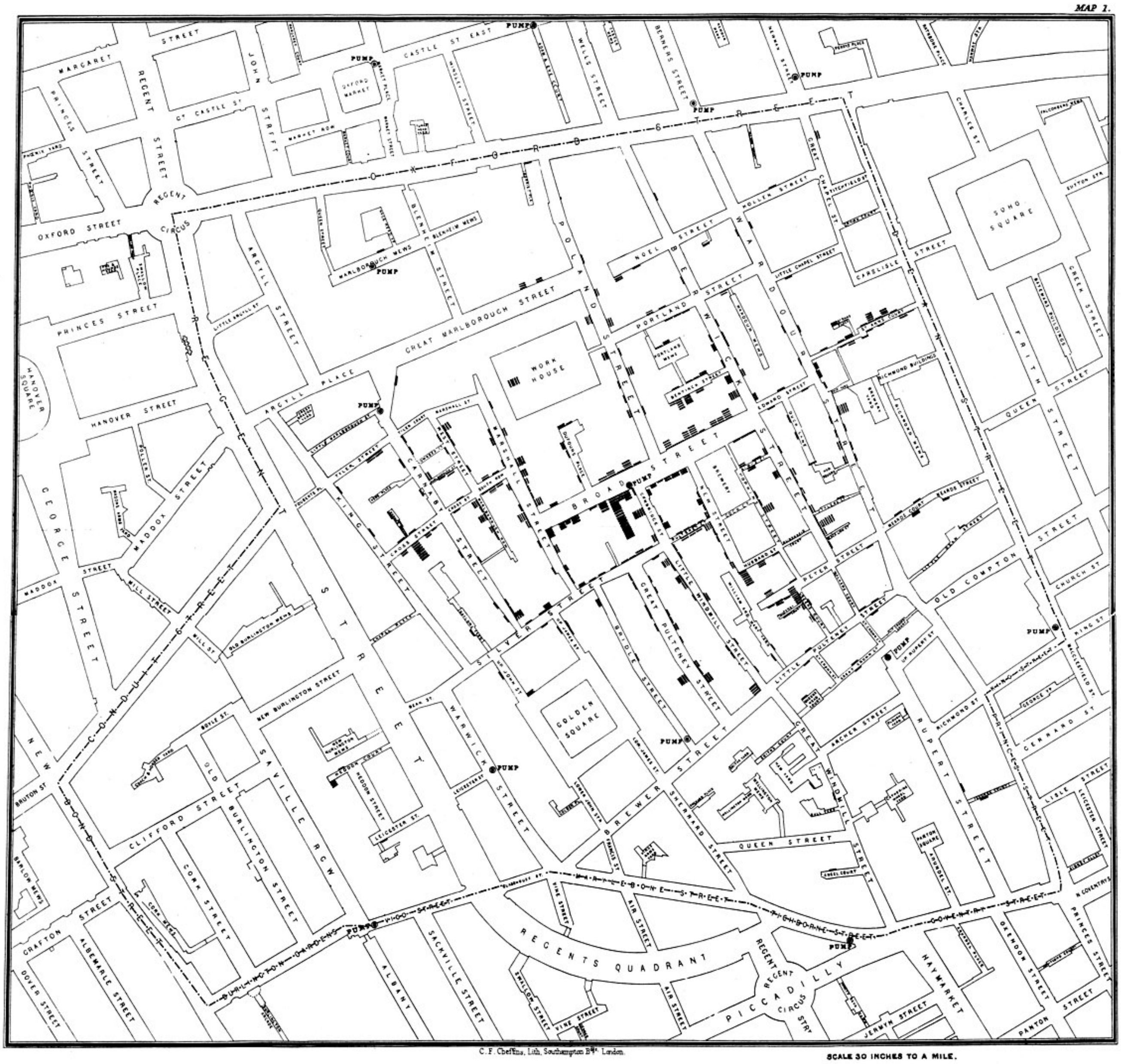


Figure 2. Map Of Cholera Outbreak

Below you can see a zoomed version of the map:

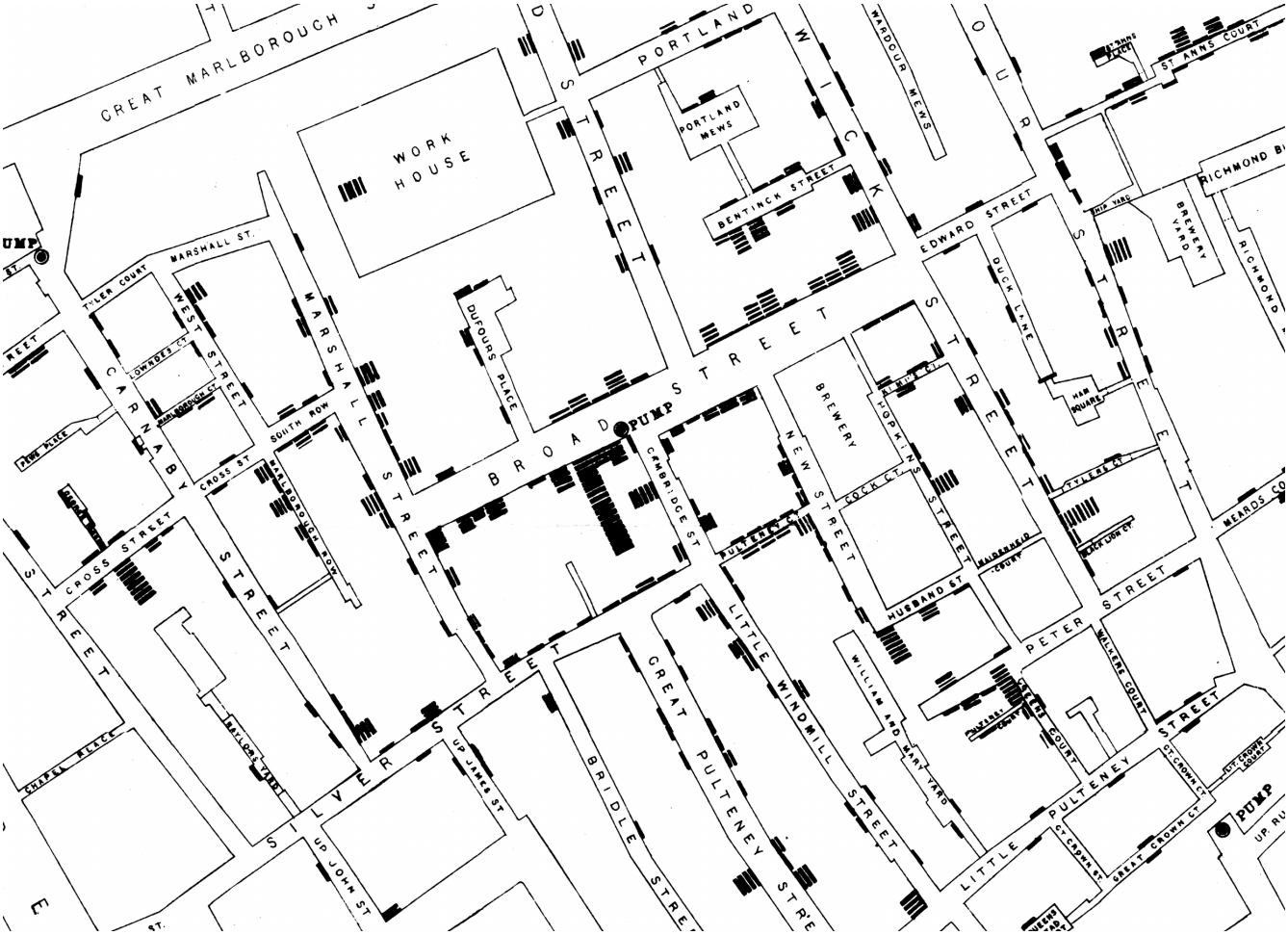


Figure 3. Zoom In On Broad Street Pump

One should be able to notice that the deaths are concentrated around Broad street pump, so John Snow concluded that the water in that pump should be contaminated.

Figure 4. Broad Street Pump

He also did his research and found out that cholera deaths that had happened quite far away from the pump were due to people drinking water from Broad street pump, as this water was brought up to them as they thought this pump water was tasty.

With the available data, and with field research, John Snow was able to show to the world that cholera is a water borne disease. With this analysis, future cholera's could be attributed / predicted to contaminated water sources, and following Snows method one could even predict where the contaminated water could be.

Here I would would like to emphasize the mapping technique, the power of human vision, which when looking at a map was able to narrow the highest amount of outbreak and was able to locate the source of cholera.

5.2. Descriptive Analysis - Napoleon’s Russian Defeat

Figure 5. French Invasion Of Russia

Napoleon Bonaparte invaded Russia. Russians did not really fight with him, but avoided him till the winter set in. The french troops were ill prepared for the Russian cold and they died in bitter and biting cold.

The French naturally wanted to study about this excellent Russian strategy and commissioned a study, and we got the following map:

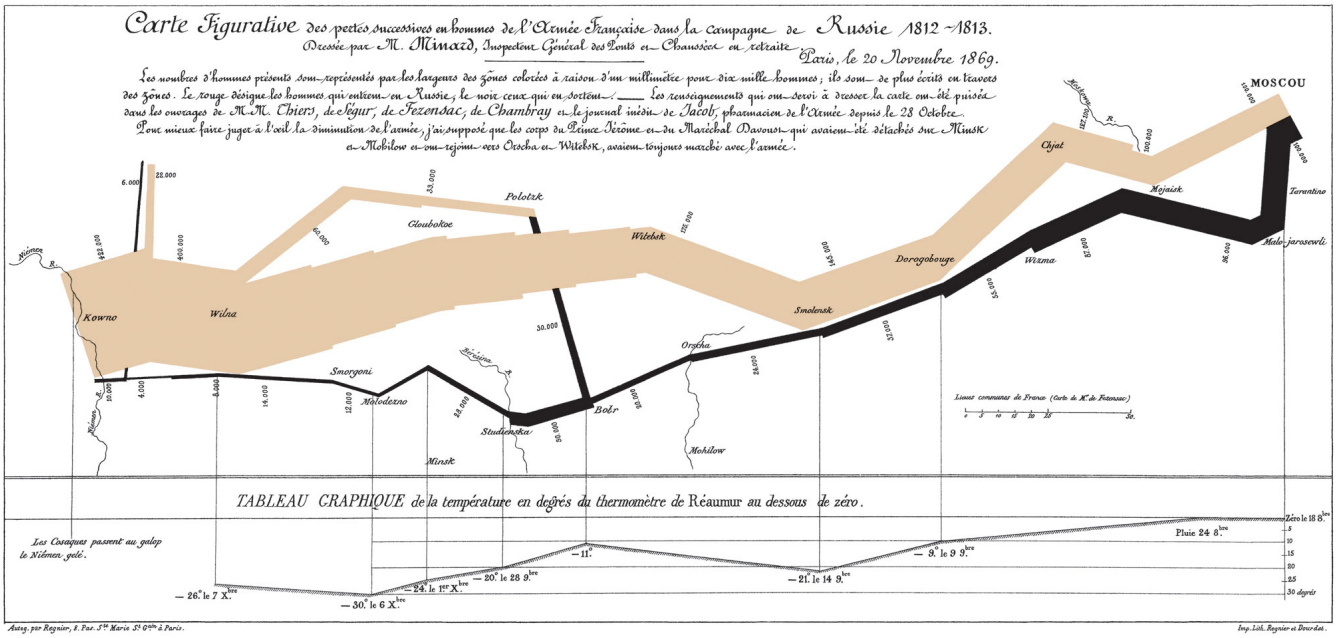


Figure 6. Troop Size Map

The map is not up to scale, but brilliantly shows the french defeat. If you take a look at it, the brown band shows the french march in to Russia, the black shows the French retreat. The width of the band shows the troop strength, so the person who is studying the visualization would know where the troop numbers reduced.

As you can see, even before the troops reached the capital Moscow, their numbers were really reduced, and the march back was disastrous. the march back is also related to a temperature chart which shows significant troop reduction along with corresponding temperatures.

Figure 7. Effect Of Low Temperature On Troops

It was finally concluded that it was not the fight with the Russians, but the low temperatures and harsh Russian winter killed most of the troops.

5.3. Prediction After Description

Certainly every time I know of, in data science assignments, it is of vital importance that a person doing the job understand the data as thoroughly as possible. So inevitably one has to do descriptive analysis even before one can predict some thing.

In the case of London cholera, it's by collecting data, mapping it, one way able to determine that it for from a well the cholera spread. The data scientist even had to find out why cholera deaths occurred even far away from the contaminated well in a case to generalize his theory. Today due to such descriptive analysis and data gathering, we know how to deal with cholera outbreak exactly and we today know it spreads by water.

Similarly, if any nation would like to invade Russia, it's better to back off if winter set's in, you simply can't beat them in winter, and their nationalism is far too higher for any army of today to defeat them.

So in order to predict cholera spreads by water, and it's better to keep ones hands off Russia, we need to analyze the data, do descriptive analysis so that we understand why things have evolved in such ways.

5.4. The Power Of Visualization

It's said that two third of our human brains processing power is dedicated to vision. I find it's true that in data science that if we could plot data, show it in visual form, them it becomes very much intuitive than looking at table of numeric values.

In the above data analysis, if you see both Joh Snow and the French have relied upon producing great visualizations to do descriptive analysis of the data presented to them, and visualization they made the understanding of data more intuitive.

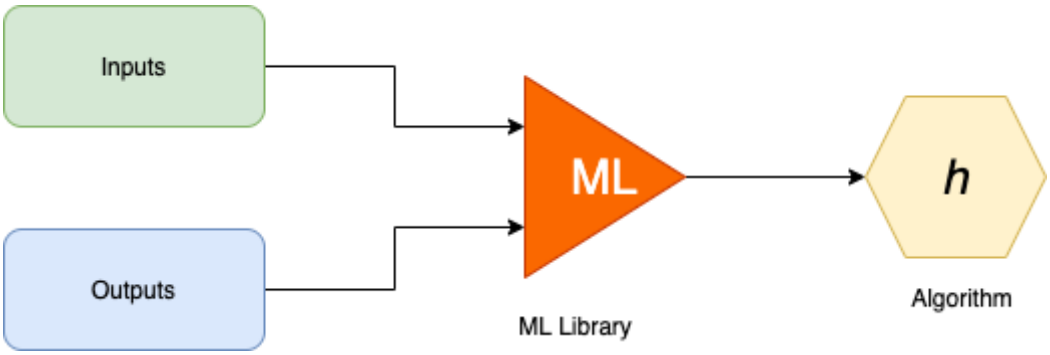
So always try to visualize data, try to create visual dashboards, this might help you to understand data science problems better and would accelerate you to solve it.

Chapter 6. Machine Learning, Artificial Intelligence and Data Science

Many people are confused about what's the difference between Machine Learning, Artificial Intelligence and Data Science. Even some who claim themselves to be professionals seems not to know how they are different, this section was created to help clarify it.

6.1. Machine Learning

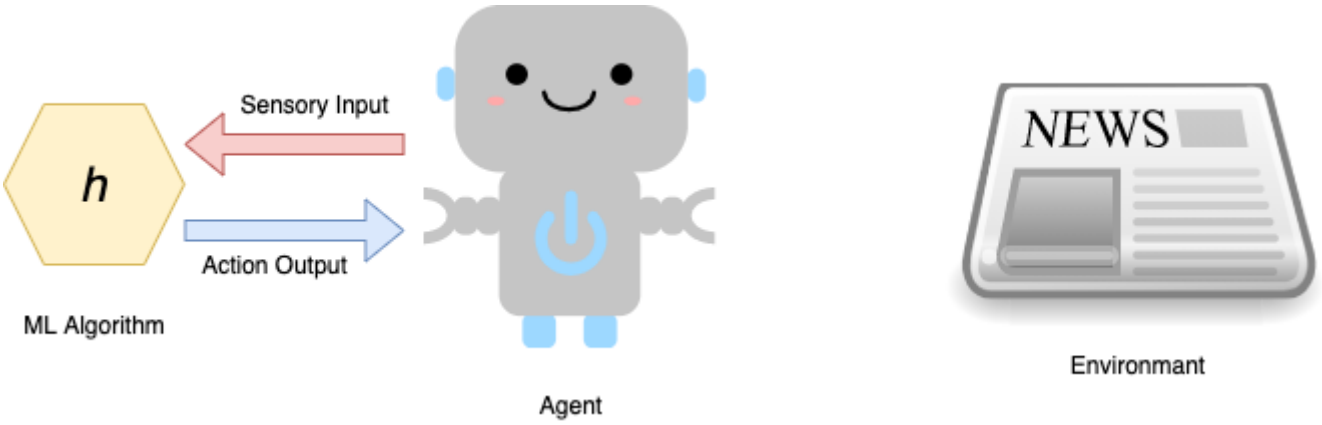
In machine learning, you would take a bunch of inputs and outputs, give it to an algorithm that would spit out a function or equation or computer code that can transform inputs to outputs.



This does not mean it just one to one mapping, that is the algorithm h is not just a table that contains the mapping from the input to output, instead it should be able to predict the values of unknown inputs as well and give a satisfactory output that varies very little compared to the real world.

6.2. Artificial Intelligence

In Artificial Intelligence you have an agent. Say you bought a robot for your house, this robot runs an algorithm that detects newspaper on your porch and brings it to you, that algorithm would have been created by machine learning process.



The robot is an agent running your machine learning algorithm, An Artificial Intelligence involves an agent that observes the real world, and acts accordingly.

Figure 8. Water Heater

One might ask that is automatic water heater is artificially intelligent, the answer is yes. You might say that the water heater operates with a bimetallic strip which isn't machine learning, my argument is so what, its able to sense the temperature, sense the set temperature, if the temperature is equal to the set temperature then it cuts off the power. It is intelligent indeed!

You don't need machine learning all the time to make something intelligent.

6.2.1. GOFAI

GOFAI stands for good old fashioned artificial intelligence where a programmer meticulously codes the rules. This might be okay for a tic-tac-toe game that plays against a human, but say an agent want's sort oranges depending on their size, texture, color and other attributes, coding that by observing the feed from a camera is going to be near impossible. Fo that GOFAI is not suitable.

So we need automated learning solutions that learns from inputs and outputs as the data becomes complex.

6.3. Data Science

Data Science is the process by which one understands data, possibly what causes the data to vary. For example, the weather man these day could tell in advance that it's going to rain, or it's going to be sunny, thats because they have poured over historical wether data patterns and had observed what causes what.

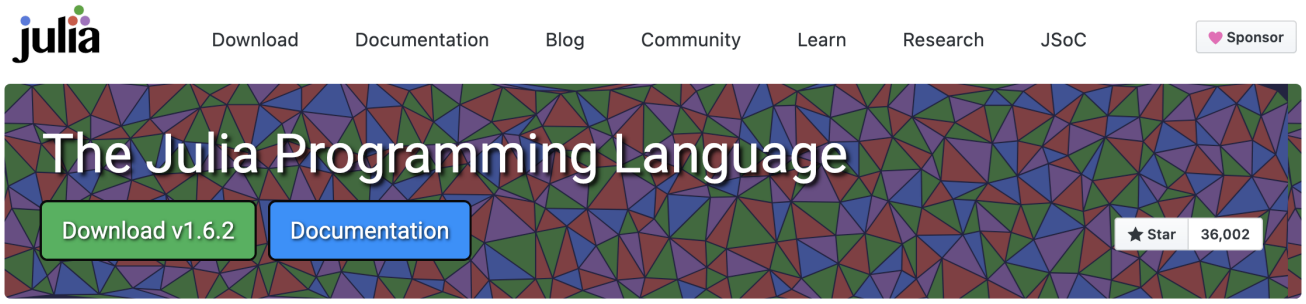
It's not necessary that a person observing things understands exactly what causes some event. For example people have long known that lunar positions caused high and low tides, but they never knew about gravity for very long time.

People in Nile have observed that certain stars appearing in certain part of the sky causes flooding of rivers, but they did not know exactly why. We humans have used our observed (data) for long before the term data science was coined. We are natural data scientists indeed. In fact even a little kid could tell whether if its mother or father would spend more on automobiles. Relating events to happenings is part of human way of life.

Julia

Chapter 7. Installing Julia

In this blog we will see how to install Julia. Visit https://julialang.org, you must see a banner like this which offers download. Click on the Download button.



You will be a taken to this page https://julialang.org/downloads/, if you scroll down you will see something like this:

Current stable release: v1.6.2 (July 14, 2021)

Checksums for this release are available in both MD5 and SHA256 formats.

Windows [help]	64-bit (installer), 64-bit (portable)	32-bit (installer), 32-bit (portable)	
macOS [help]	64-bit		
Generic Linux on x86 [help]	64-bit (GPG), 64-bit (musl)[1] (GPG)	32-bit (GPG)	
Generic Linux on ARM [help]	64-bit (AArch64) (GPG)	32-bit (ARMv7-a hard float) (GPG)	
Generic Linux on PowerPC [help]	64-bit (little endian) (GPG)		
Generic FreeBSD on x86 [help]	64-bit (GPG)		
Source	Tarball (GPG)	Tarball with dependencies (GPG)	GitHub

I clicked on the 64-bit thing for Generic Linux x86. Once you do that a file named `julia-1.6.2-linux-x86_64.tar.gz` will be downloaded. Extract it to your home directory and open `~/.bashrc` and add it this at the last:

```
# Add Julia to PATH
export PATH="$PATH:$HOME/julia-1.5.2/bin"
```

this tells your computer to look into `~/julia-1.6.2/bin` when you call `julia`. Just for once type this in terminal:

```
$ source ~/.bashrc
```

To know more about `.basrc` you can visit here https://www.journaldev.com/41479/bashrc-file-in-linux.

In your terminal when you type `julia`, you should be getting this:

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.5.2 (2020-09-23)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |

julia>
```

Press `Ctrl+D` to exit.

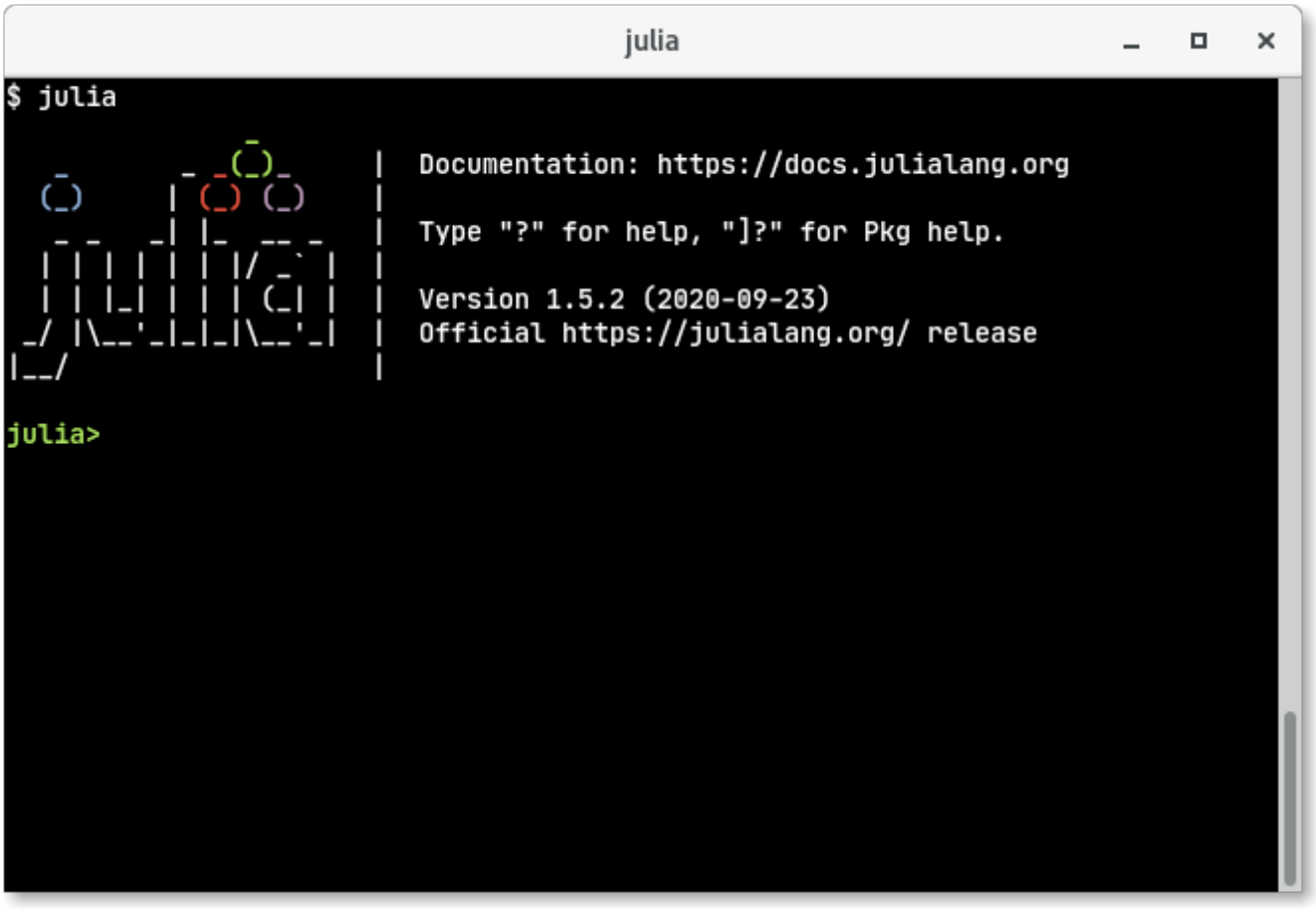
If all had gone well, you had installed Julia, ran its REPL, and came out of it successfully. In case you are wondering what REPL is, wait for the next blog.

Chapter 8. Julia REPL

REPL stands for Read Eval(uate) Print Loop. Its a way a programming ecosystem gives you a easy way to execute small programs in shell environments. Let's get started with it. In your terminal type this:

```
$ julia
```

You should be seeing something like this.



Juila prints a nice ASCII art of Julia logo and it prints version info and when was it released etc. The julia> is julia prompt where you can type some small Julia code, Julia will read it, evaluate it and print the result.

So lets see what 1 and 2 added is, type `1 + 2` and hit ENTER:

```
julia> 1 + 2
3
```

So it's 3 yaaay! Our first small Julia program!!

You can type multiple Julia statements per line, it needs to be separated by semicolon `;` as shown:

```
julia> x = 3; y = 4
4
```

In the above statement we have a thing like `x = 3`, this means we put a value of 3 in a box named `x` and in `y = 4`, we put 4 in a box named `y`. Now what happens when we have three x boxes and four y boxes? Try this out:

```
julia> 3x + 4y
25
```

Julia says its 25!

Now let's try to multiply `x` and `y`:

```
julia> xy
ERROR: UndefVarError: xy not defined
```

It fails. We will see why it fails when we see about variables. Okay, mathematically we use dot . to denote multiplication

ion, so let's try out `x.y`:

```
julia> x.y
ERROR: type Int64 has no field y
Stacktrace:
 [1] getproperty(::Int64, ::Symbol) at ./Base.jl:33
 [2] top-level scope at REPL[5]:1
```

it fails too, we will soon see why in coming blogs (i.e you will learn about it). The right way to multiply in Julia is to use the star * operator as shown:

```
julia> x * y
12
```

8.1. Volume of Sphere

Let's see something complex. Let's now calculate volume of a sphere. We know its $\frac{4}{3}\pi r^3$, lets see how to do it in Julia.

First let's have a variables name `radius` to hold value of radius, so let's say the radius is 7 units, so we use this statement to assign 7 to radius:

```
julia> radius = 7
7
```

Next we calculate the volume as follows:

```
julia> (4/3)π * radius^3
1436.7550402417319
```

If you are wondering how I got π into Julia REPL, just type `\pi` and press kbd:[TAB]. Here radius^3 means radius is raised to the power of 3.

Julia provides a function named `typeof`, with which we can investigate the type of π as follows:

```
julia> typeof(π)
Irrational{:π}
```

so Julia says π is irrational which is right.

Rational numbers in Julia can be defined like `x//y`, so `typeof(22//7)`` returns rational as shown:

```
julia> typeof(22//7)
Rational{Int64}
```

8.2. Clearing REPL

When you feel there are lot of things in your Julia REPL and want to have a fresh screen, just type Ctrl + l and it will blank out as shown:

You can still access old variables and functions defined, its just the display that's blanked out.

8.3. Exiting a statement

When you have typed something wrong and want to terminate the statement, just type kbd:[Ctrl + c].

```
julia> 1 + 41^C

julia>
```

Julia will clear out that statement and present a fresh prompt.

8.4. History

Use the up arrow and down arrow to go through the history of all the statements you have typed.

8.5. Exiting REPL

To exit Julia REPL press kbd:[Ctrl + d].

Chapter 9. Accessing Help

Video lecture for this section could be found here https://youtu.be/BC11KlSQm6E

If you have been using GNU/Linux or Python (even on on evil platforms like Mac and Windows), you will have a way of accessing help documentation in REPL, so does Julia too. To access help, just type ? in Julia prompt, you will be presented with something like this:

```
help?>
```

You can come out of help by pressing kbd:[Backspace]. Okay lets see what typeof (if you have read previous blog you may have used typeof) does. Just type typeof in help prompt and hit Enter.

```
help?> typeof
search: typeof typejoin TypeError

  typeof(x)

  Get the concrete type of x.

  Examples
  ⋅⋅⋅⋅⋅⋅⋅⋅⋅⋅

  julia> a = 1//2;

  julia> typeof(a)
  Rational{Int64}

  julia> M = [1 2; 3.5 4];

  julia> typeof(M)
  Array{Float64,2}
```

Isn't it useful?

Chapter 10. Package Management

Video for this part of the book https://youtu.be/6_pcQB9n9jI

et's see how to add packages in Julia using it's REPL. For that just like you went into help mode by pressing `?` at `julia>` prompt, you must for package management press `]`. To come out of this package `pkg>` thing press `Backspace`.

In case I am going too fast, let's discuss what packages are. Today is the age of internet, and hence a code written by one can be used by millions, all it needs is a internet connection. So almost all modern languages have a way of managing reusable shared code. In Julia, this reusable shared code is called a Package.

Since Julia's ecosystem is open source (I wish people understood the merits of [Free Software](https://fsf.org) and ignored the misleading Open Source, any way), many people look at popular packages in Julia and analyze them, they are well maintained and tested and can used with fair amount of trust. So let's see how to install a package.

10.1. Installing packages

So to make our learning useful, let's install a package that highlights out Julia code typed in the Julia prompt. So press `]` and type `add OhMyREPL` and press `ENTER`, this will spit some output as shown:

```
(@v1.5) pkg> add OhMyREPL
 Installing known registries into `~/.julia`
################################################################################ 100.0%
      Added registry `General` to `~/.julia/registries/General`
  Resolving package versions...
  Installed Pipe ──────────── v1.3.0
  Installed Tokenize ──── v0.5.8
  Installed OhMyREPL ──── v0.5.9
  Installed Crayons ────── v4.0.4
  Installed fzf_jll ────── v0.21.1+0
  Installed JLFzf ─────────── v0.1.2
Downloading artifact: fzf
Updating `~/.julia/environments/v1.5/Project.toml`
  [5fb14364] + OhMyREPL v0.5.9
Updating `~/.julia/environments/v1.5/Manifest.toml`
  [a8cc5b0e] + Crayons v4.0.4
  [1019f520] + JLFzf v0.1.2
  [5fb14364] + OhMyREPL v0.5.9
  [b98c9c47] + Pipe v1.3.0
  [0796e94c] + Tokenize v0.5.8
  [214eeab7] + fzf_jll v0.21.1+0
  [2a0f44e3] + Base64
  [ade2ca70] + Dates
  [b77e0a4c] + InteractiveUtils
  [76f85450] + LibGit2
  [8f399da3] + Libdl
  [56ddb016] + Logging
  [d6f4376e] + Markdown
  [44cfe95a] + Pkg
  [de0858da] + Printf
  [3fa0cd96] + REPL
  [9a3f8284] + Random
  [ea8e919c] + SHA
  [9e88b42a] + Serialization
  [6462fe0b] + Sockets
  [cf7118a7] + UUIDs
  [4ec0a83e] + Unicode
```

Hope all went well, now press `Backspace` to come back to Julia prompt

```
julia>
```

Now press `Ctrl + d` to come out of Julia.

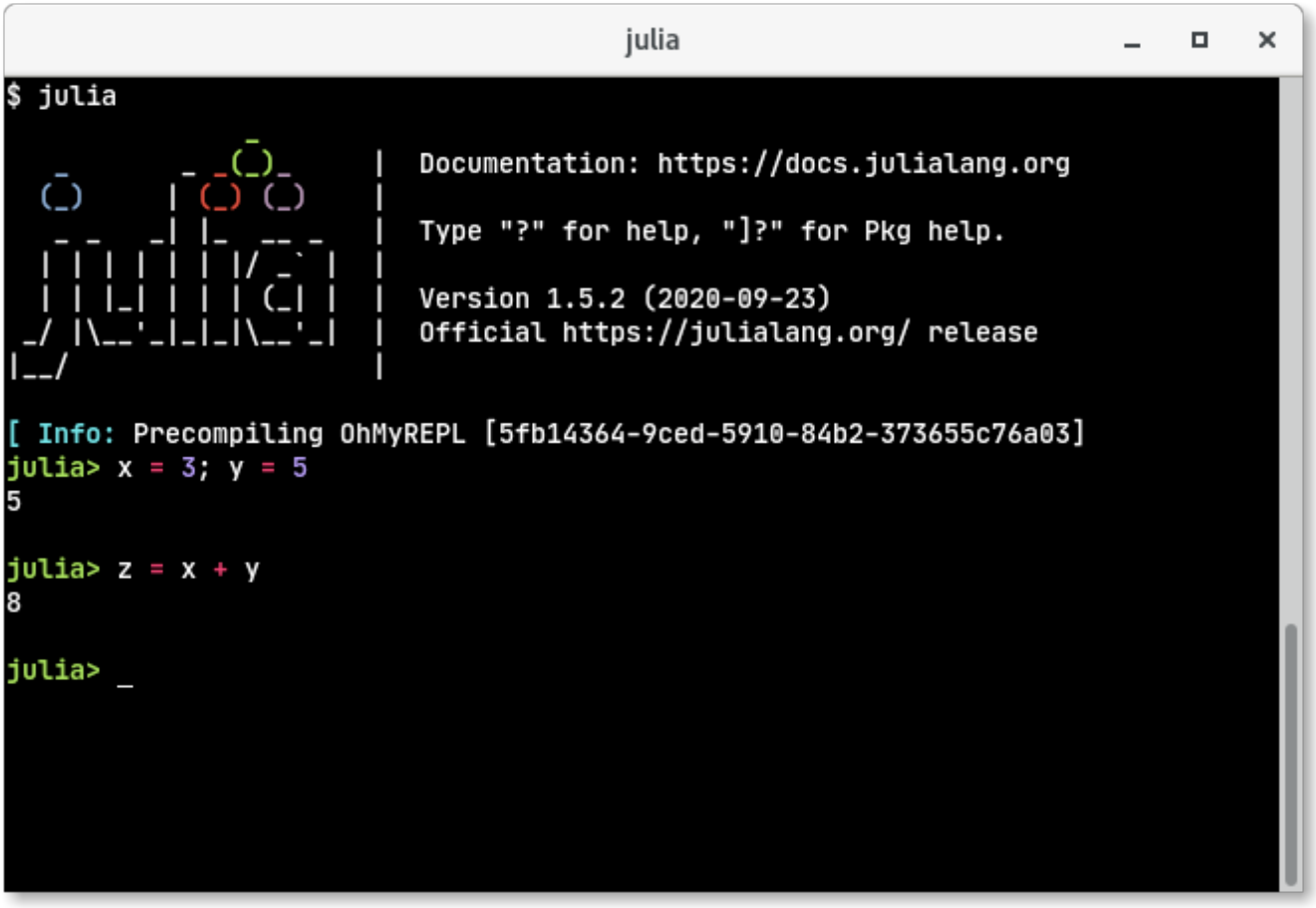
Open a editor, open the file `~/.julia/config/startup.jl`, if it's not present create it, and add this code in it:

```
atreplinit() do repl
    try
        @eval using OhMyREPL
    catch e
        @warn "error while importing OhMyREPL" e
    end
end
```

The above code tells to include the package `OhMyREPL` during startup of `julia>` prompt. Save the file. Now in terminal type:

```
$ julia
```

to start Julia REPL, and since this is the first time you are using `OhMyREPL`, julia will compile it and you will see something as shown below:



So when you try out expressions as shown above, it's color highlighted! Enjoy!!!

10.2. Removing packages

To remove a package, you need to do this

```
pkg> rm PackageName
```

10.3. Reference

- Julia Packages https://docs.julialang.org/en/v1/stdlib/Pkg/
- OhMyREPL https://juliapackages.com/p/ohmyrepl

Chapter 11. Installing Jupyter notebook and Jupyter lab

Video lecture for this section could be found here https://youtu.be/sJUrKTCYM64

Data Science is a highly interactive activity. You get the data, do something with it, see what has happened, then go a step back or step forward depending on the result and so on. Usually data scientist don't use REPL and a text editor to code, they use something interactive called Jypyter[8] notebooks which you will learn about in the coming, chapters. Over here we will see how to install it.

11.1. Install IJilia

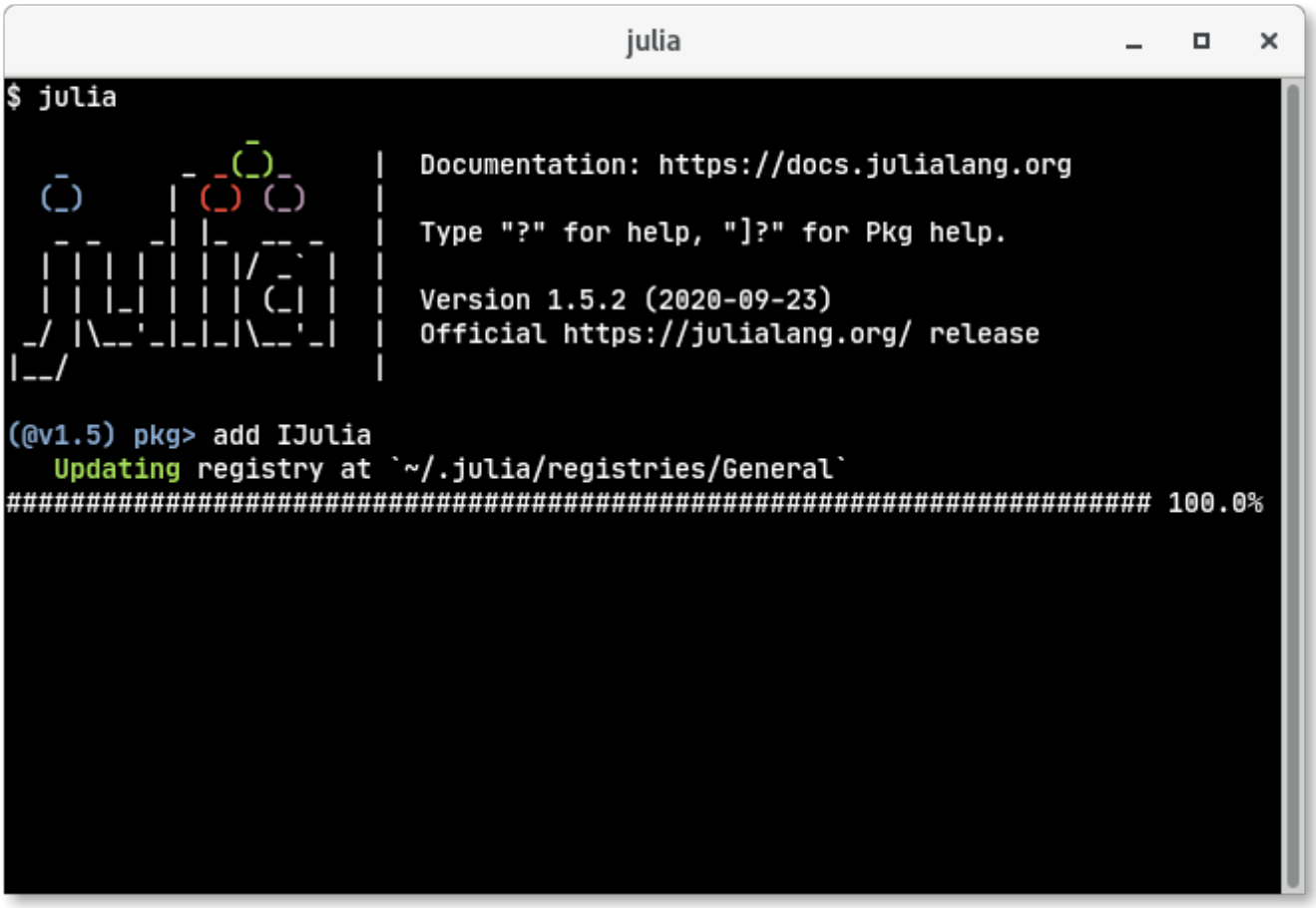
To install IJulia, the package that let's Julia connect with Jupyter. First launch Julia REPL by executing this:

```
$ julia
```

Once again you will greeted by Julia's ASCII art screen

```
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.5.2 (2020-09-23)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/                   |
```

Goto packages by pressing `]` and type `add IJulia` at the `pkg>` prompt. You must see something like this



This was the thing that `pkg>` spit out for me

```
(@v1.5) pkg> add IJulia
   Updating registry at `~/.julia/registries/General`
######################################################################## 100.0%
  Resolving package versions...
  Installed Artifacts ──────────── v1.3.0
  Installed MbedTLS_jll ────────── v2.16.8+1
  Installed VersionParsing ─── v1.2.0
  Installed ZeroMQ_jll ─────────── v4.3.2+5
  Installed ZMQ ────────────────────── v1.2.1
  Installed MbedTLS ────────────────── v1.0.3
  Installed Parsers ────────────────── v1.0.11
  Installed IJulia ─────────────────── v1.22.0
  Installed SoftGlobalScope ── v1.1.0
  Installed Conda ──────────────────── v1.4.1
  Installed JSON ───────────────────── v0.21.1
  Installed JLLWrappers ────────── v1.1.3
Downloading artifact: ZeroMQ
Downloading artifact: MbedTLS
Updating `~/.julia/environments/v1.5/Project.toml`
  [7073ff75] + IJulia v1.22.0
Updating `~/.julia/environments/v1.5/Manifest.toml`
  [56f22d72] + Artifacts v1.3.0
  [8f4d0f93] + Conda v1.4.1
  [7073ff75] + IJulia v1.22.0
  [692b3bcd] + JLLWrappers v1.1.3
  [682c06a0] + JSON v0.21.1
  [739be429] + MbedTLS v1.0.3
  [c8ffd9c3] + MbedTLS_jll v2.16.8+1
  [69de0a69] + Parsers v1.0.11
  [b85f4697] + SoftGlobalScope v1.1.0
  [81def892] + VersionParsing v1.2.0
  [c2297ded] + ZMQ v1.2.1
  [8f1865be] + ZeroMQ_jll v4.3.2+5
  [8ba89e20] + Distributed
  [7b1f6079] + FileWatching
  [a63ad114] + Mmap
  [8dfed614] + Test
   Building Conda ──→ `~/.julia/packages/Conda/3rPhK/deps/build.log`
   Building IJulia → `~/.julia/packages/IJulia/a1SNk/deps/build.log`

(@v1.5) pkg>
```

Exit `pkg>` by pressing `backspace`.

11.2. Start Jupyter Notebook

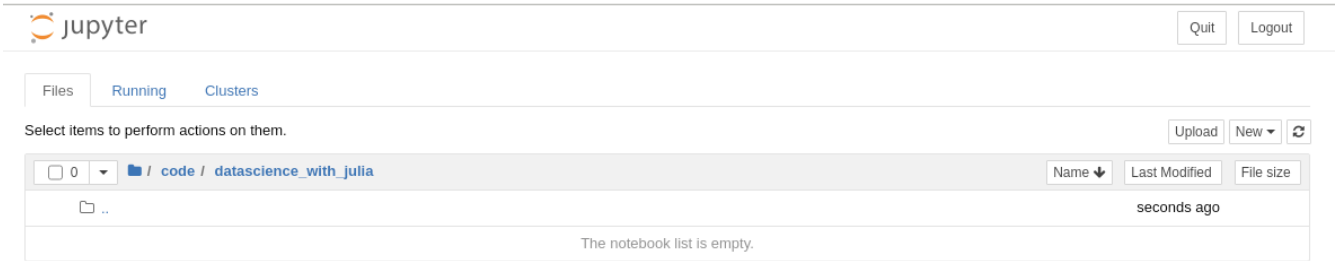
Now at your `julia>` prompt type `using IJulia` as shown below:

```
julia> using IJulia
[ Info: Precompiling IJulia [7073ff75-c697-5162-941a-fcdaad2a7d2a]
```

If this is your first time initiating IJulia, then it will precompile it as shown above, now type `notebook()` as shown below:

```
julia> notebook()
[ Info: running `/home/karthikeyan/anaconda3/bin/jupyter notebook`
^CProcess(setenv(`/home/karthikeyan/anaconda3/bin/jupyter notebook`; dir
="/home/karthikeyan"), ProcessExited(0))
```

This should open the notebook in the browser, and you must see something like this.



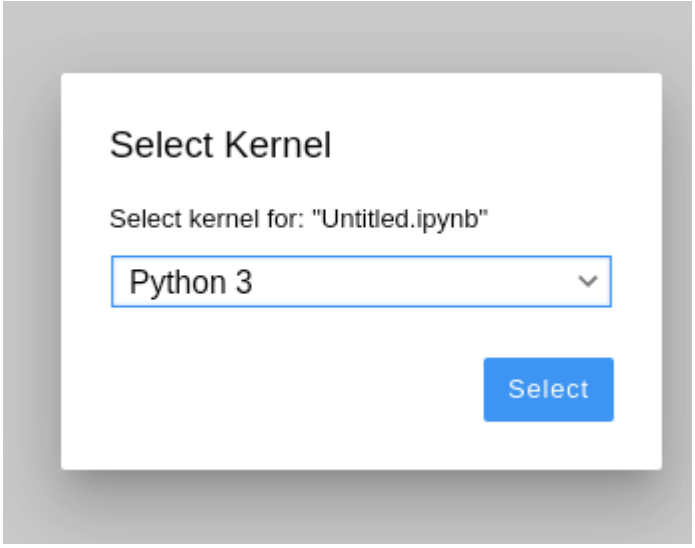
To quit notebook type `Ctrl + c` in the terminal.

11.3. Start Jupyter Lab

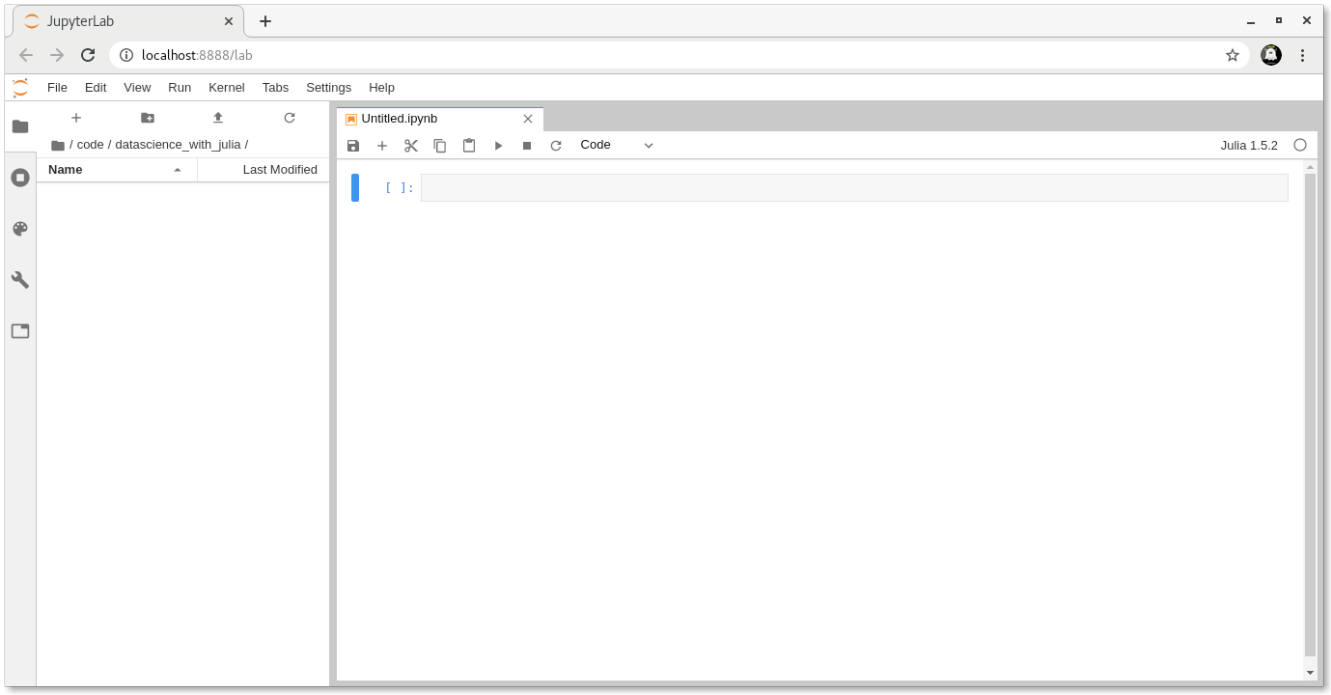
To start Jupyter Lab, type the following commands at the `julia>` prompt:

```
julia> using IJulia
julia> jupyterlab()
```

Your browser will launch and you will see a prompt that asks you to select the language that Juptyter lab shold operate on, I selected Julia 1.5



You will see a screen like this for Jupyter lab



To stop Jupyter lab type `Ctrl + c` at the terminal. In the next few blogs we will see the basics of notebook and lab.

11.4. Reference

- https://jupyter.org/install
- https://github.com/JuliaLang/IJulia.jl

[8] https://jupyter.org/

Chapter 12. Starting with Julia (using Jupyter) lab

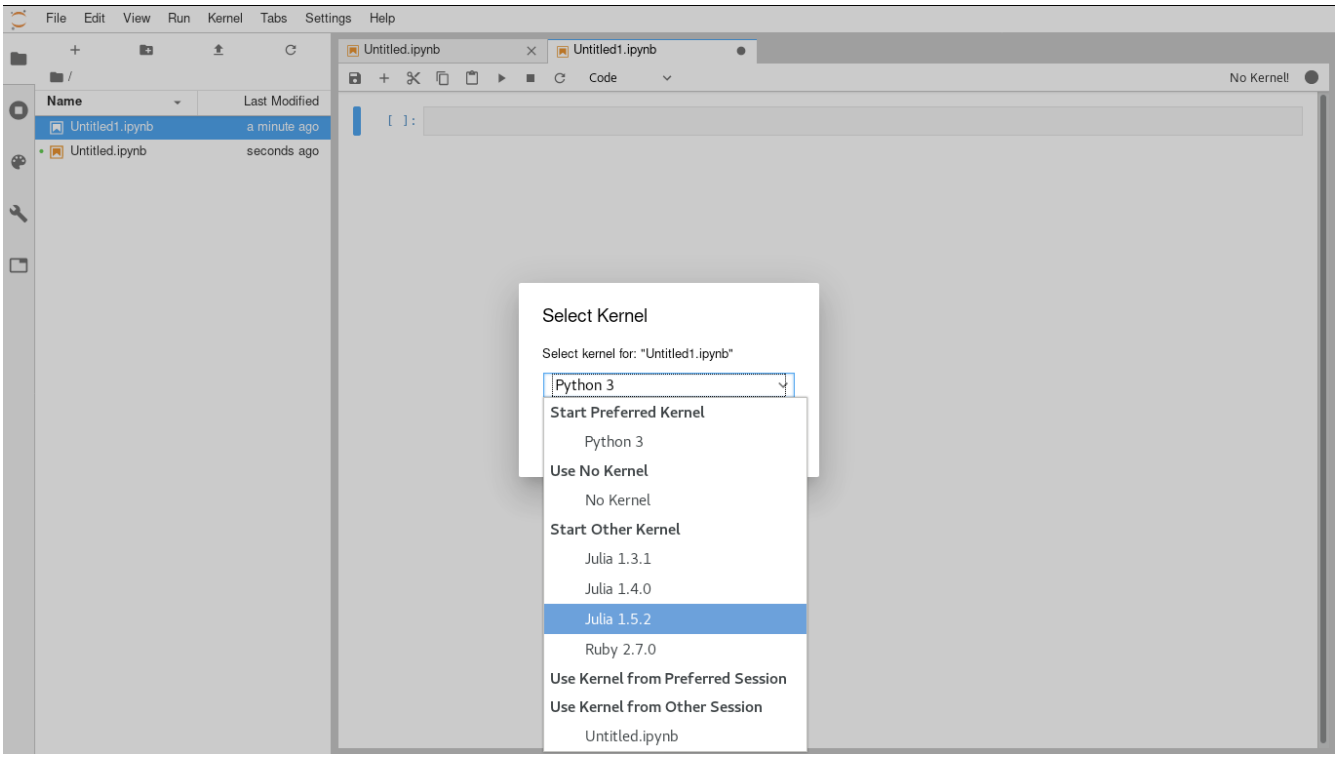
NOTE

Get video lecture for this section here https://youtu.be/aaMsJawnJuQ

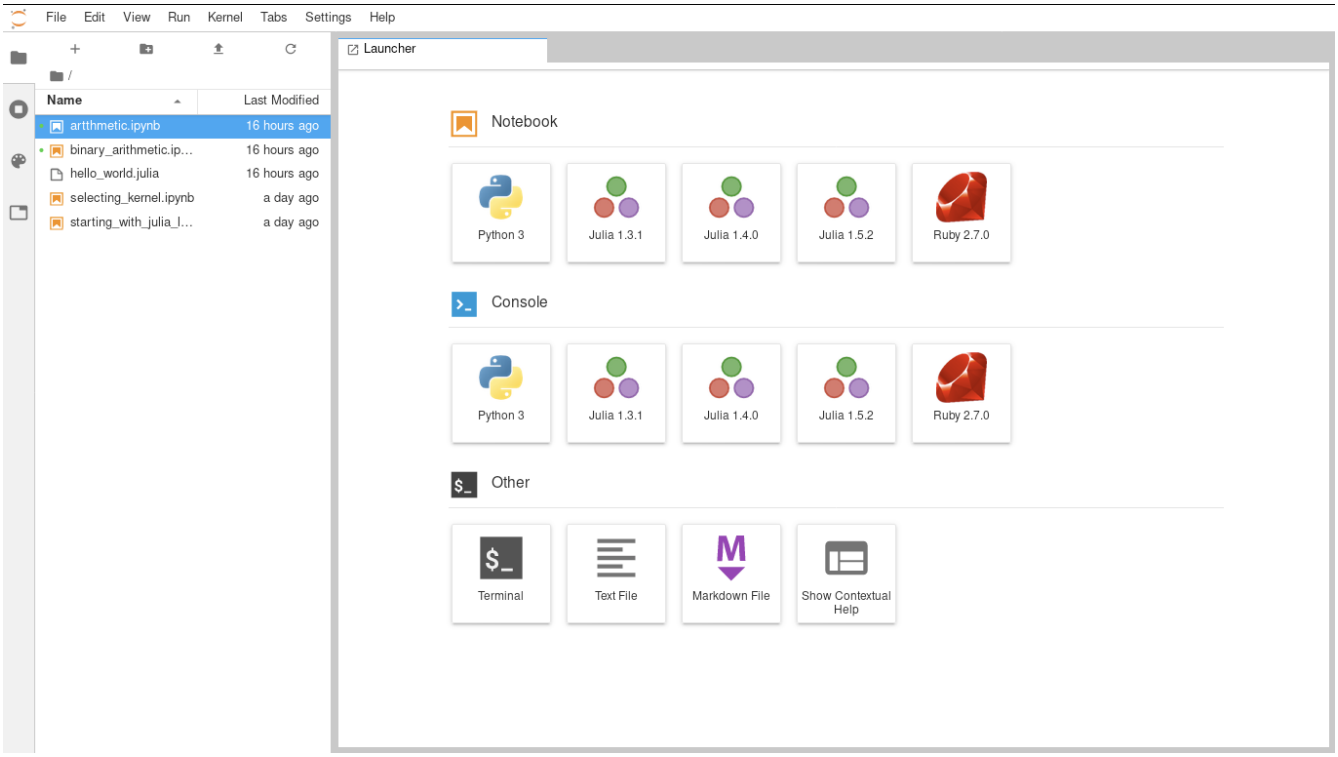
Okay, let's start with Juyter lab. I am using Jupyter notebooks no more because I feel lab is cool. To start it you can either do it by the Julia way by calling IJulia as I mentioned [Installing Jupyter notebook and Jupyter lab]({% post_url 2020-10-31-installing-jupyter-notebook-and-jupyter-lab %}), or I like the Conda way, to do it, in your terminal type this:

```
$ jupyter lab
```

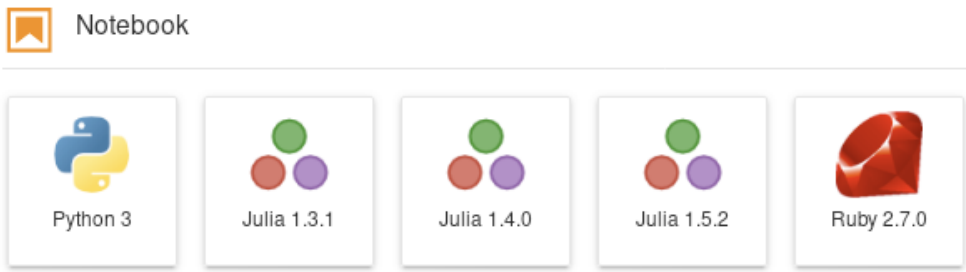
When started in IJulia way, the lab asks me what kernel I must use as shown below:



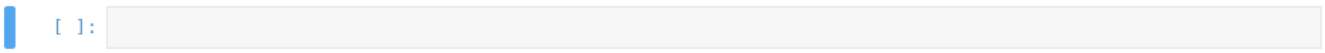
when started the Conda way, it sows something like this:



In the above launcher, I select Julia 1.5.2 in the Notebook section:



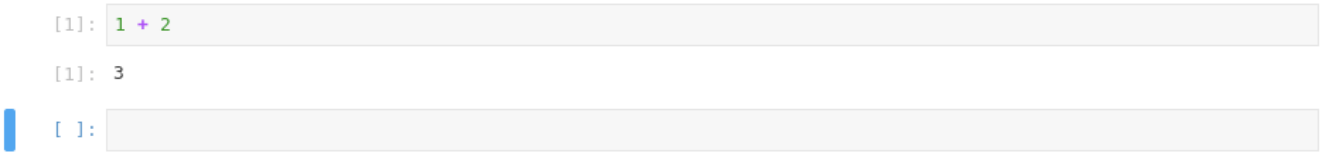
When you select a kernel, at the left pane you should be seeing a file called `untitled.ipynb`, right click on it and change it's name to `starting_with_julia_lab.ipynb`. Now this lab is like REPL on steroids, it has got thing called cells as shown:



Now type

```
1 + 2
```

in the cell and press `Shift + Enter`, you see the outputs `3` and the input cell receives a number `[1]:`, the output too receives number `[1]` as shown in the image below



Now let's try to print the legendary [Hello World](https://en.wikipedia.org/wiki/

%22Hello,_World!%22_program) in the next cell

```
println("Hello World")
```

It should spit out an output as shown:

```
Hello World
```

Now let's print `1 + 2 = 3`, for that type the following in the cell

```
println("1 + 2 = ", 1 + 2)
```

and press `Shift + Enter` or `Ctrl + Enter`

```
1 + 2 = 3
```

You will get an output as shown above

in the coming example, we put a value `3` into a thing called variable, this variable is called `a` and you know how to execute this, go on:

```
a = 3
```

Output:

```
3
```

Similarly we put a value `5` in variable `b` and execute it:

```
b = 5
```

Output:

```
5
```

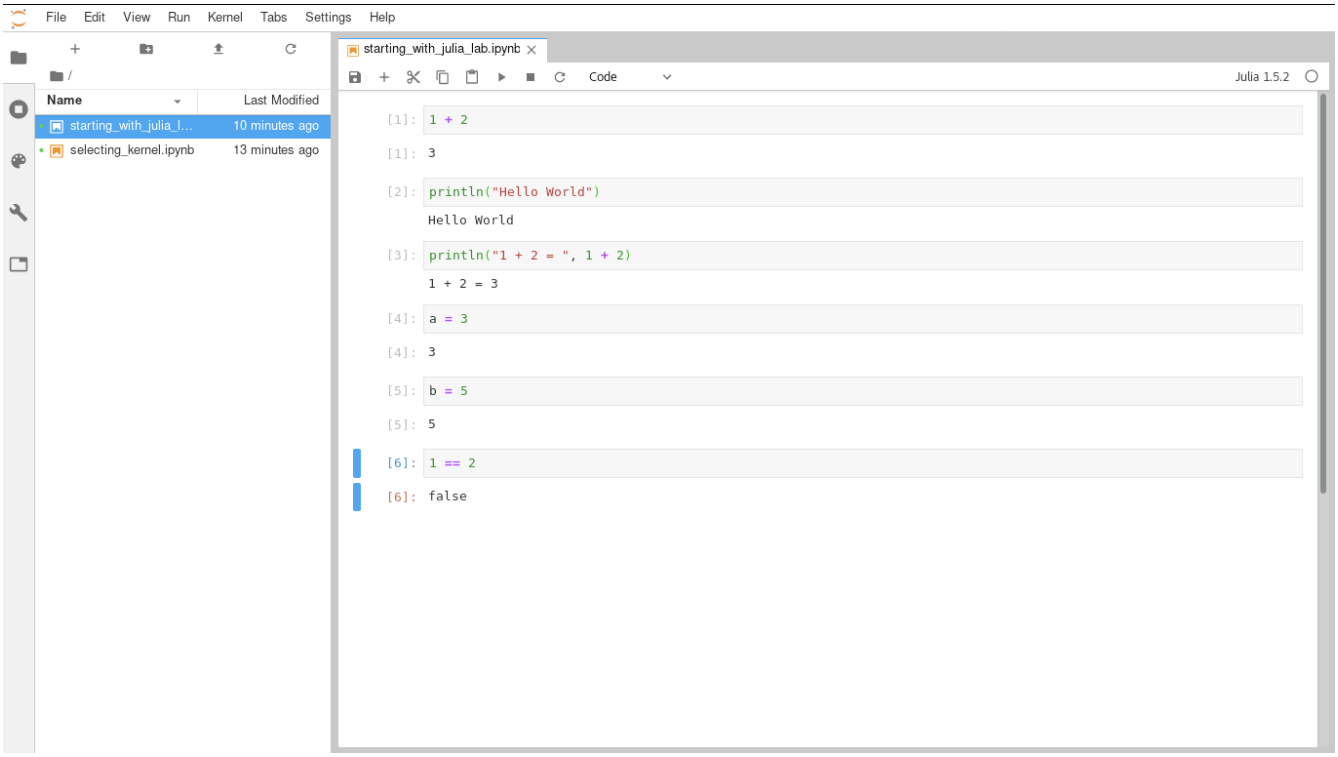
Now let's check if 1 equals 2 in the next cell:

```
1 == 2
```

Output:

```
false
```

Once you have done this your notebook should look like this.



A Jupyter lab instance is collection of many notebooks. If you did not understand what I mean by that, its okay, things will clear up.

Instead of `Shift + Enter`, you can press `Alt + Enter`, in that case the cell will be executed, and a new cell won't be created beneath it. To get the code / note book for this blog check out this link https://gitlab.com/data-science-with-julia/code/-/blob/master/starting_with_julia_lab.ipynb

Go on and play around with Jupyter lab, mess it up and crash it, start it all over to learn.

Chapter 13. Julia program in a file

Video lecture for this section could be found here https://youtu.be/_oDSlQuny4Y

It's not that all your Julia programs will be in Jupyter Lab and Jupyter notebooks. Once you have done some thing and its kinda set in stone, you can put in a file and and execute it using the Julia run time. In production environment, its nothing but few Julia files getting executed when needed. In this blog we will see how to put Julia programs in a file and execute it.

I hope you have a text editor with you, I strongly do not recommend using Juptyer lab to create Julia files. Why? Because I don't like it and you will see for yourself if you don't realize it now. I would suggest one to use [Atom](https://atom.io) or [VS Code](https://code.visualstudio.com/), unfortunately both created by evil Microsoft corporation.

Create a file called `hello_world.julia` and put this stuff into it:

```
println("Hello World!")
println("Let's see what miracles the new field of Data Science can do.")
```

you can get this source code here https://gitlab.com/data-science-with-julia/code/-/blob/master/hello_world.julia

Now in your terminal, go to the directory where the file is and type this:

```
$ julia hello_world.julia
```

and hit `Enter`. If you get this:

```
Hello World!
Let's see what miracles the new field of Data Science can do.
```

as output, then all is well. Yaay, you have created and executed your first Julia program as a file. In data science we explore a lot with notebooks and lab because of their visual nature. When things are okay, we put them in Julia files and ship it off.

Chapter 14. Basic Arithmetic

Video lecture for this section could be found here https://youtu.be/4g0eImxZ_K4

Let us see about basic arithmetic in Julia. The notebook for this blog is available here https://gitlab.com/datascience-book/code/-/blob/master/artthmetic.ipynb.

First start a notebook, so first let's do addition. Type this in a cell:

```
1 + 2
```

and execute it. As you can see + sign adds two numbers and outputs the result as shown:

```
3
```

Try an other example:

```
41 + 1
```

Output:

```
42
```

Now let's try multiplication:

```
6 * 7
```

The * sign multiplies numbers at its left and right side and outputs the result as shown:

```
42
```

Now let's try subtraction which is accomplished by - sign. In the example below we are taking away 8 from 50:

```
50 - 8
```

Output:

```
42
```

Now division is done by /, so this is what you get when you divide 375 by 21:

```
375 / 21
```

Output:

```
17.857142857142858
```

Now x^y in Julia can be written as `x ^ y`, so the below example finds out 28 raised to the power of 12:

```
28 ^ 12
```

Output:

```
232218265089212416
```

For curiosity reason we will see what will happen when we add -2 and 45:

```
-2 + 45
```

Output:

```
43
```

There is a operator called mod % which gives us the reminder. Lets see what's the reminder when 21 is divided by 4:

```
21 % 4
```

Output:

```
1
```

We have seen how to get reminder, now let's see how to get quotient of a division operation. For that we use ÷ operator which does integer division. So look at the example below:

```
21 ÷ 4
```

Output

```
5
```

To type in ÷ in the notebook, type `\div` in a cell and press `Tab`.

It's not that one can enter only 1 statement in Jupyter lab's cell and press `Shift + Enter` to execute it, you can enter multiple statements by using `Enter` key at the end of statement, then you can press `Shift + Enter` to execute it.

In the example below, we convert 90 degree Celsius to Fahrenheit.

```
celcius = 90
fahrenheit = (9 / 5)celcius + 32
```

Output:

```
194.0
```

In the above example, first we assign value `90` to variable named `celcius` using this statement `celcius = 90`. Next we compute Fahrenheit as follows `fahrenheit = (9 / 5)celcius + 32`. Look how we can give `(9 / 5)celcius`, Julia can find out you are trying to multiply $\frac{9}{5}$ with `celcius`, it's very much like writing an equation. Then `32` is added to it here `(9 / 5)celcius + 32`.

For the sake of experimentation try out this statement `fahrenheit = (9 / 5) * (celcius + 32)`. Did you get the right answer? What went wrong?

Now try out the example below:

```
celcius = 90
fahrenheit = (9 / 5) celcius + 32
```

Output:

```
syntax: extra token "celcius" after end of expression



Stacktrace:

 [1] top-level scope at In[14]:2

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

Note how quickly Julia could get confused by just leaving a space between `(9/5)` and `celcius`, wheras you human can see its a obvious thing.

Now try this example out:

```
celcius = 90
fahrenheit = (9 / 5) * celcius + 32
```

Output:

```
194.0
```

There is a difference between how Scientists write equations and programmers program it. For example for a scientist writing `F = (9/5)C + 32` is a good enough Julia program, but as a programmer one would absolutely hate it. A programmer would like the program to be self documenting. Hence he would like `fahrenheit = (9 / 5) * celcius + 32`. He would like to mention `celcius` is multiplied by `(9 / 5)` explicitly using the `*` operator.

For a scientist `(9/5)` is good enough, but for a programmer its blasphemy. Programmers like to the program to be more readable. I think I need to write a blog about the way of code: Scientists Vs Programmers. `(9 / 5)` is more readable than `(9/5)` programmer would say.

Now consider the example below. When ever you wrap something in brackets, it get's executed first, so in the below operation `90` is added with `32` in `(celcius + 32)` giving `122`, then this `122` is multiplied with 9 \over 5 in `(9 / 5) * (celcius + 32)`:

```
celcius = 90
farenheit = (9 / 5) * (celcius + 32)
```

Output:

```
219.6
```

You can use only round brackets `(` and this `)` in math iperations in Julia, curly and other brackets are strict no:

```
celcius = 90
fahrenheit = {9 / 5} * celcius + 32
```

Output:

```
syntax: { } vector syntax is discontinued around In[17]:2



Stacktrace:

 [1] top-level scope at In[17]:2

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

Here I am trying to use square brackets:

```
celcius = 90
fahrenheit = [9/5] * celcius + 32
```

Output:

```
MethodError: no method matching +(::Array{Float64,1}, ::Int64)
For element-wise addition, use broadcasting with dot syntax: array .+ scalar
Closest candidates are:
  +(::Any, ::Any, !Matched::Any, !Matched::Any...) at operators.jl:538
  +(!Matched::Missing, ::Number) at missing.jl:115
  +(!Matched::Base.CoreLogging.LogLevel, ::Integer) at logging.jl:116
  ...



Stacktrace:

 [1] top-level scope at In[18]:2

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

One good thing about Julia is you can assign values to variables and construct algebric equations as follows:

```
x = 4
y = 7

3x + 4y + 27
```

Output:

```
67
```

Try breaking the above example and make it throw errors so that you would learn.

Let's assign `12` to `z` as shown:

```
z = 12
```

Output:

```
12
```

Below example is the compact way of writing `z = z + 4`, that is you are adding `4` with `z`, making it `16` and assigning it to `z` again.

Example 1:

```
z += 4
z
```

Output:

```
16
```

Imagine what would a mathematician think if you present an equation z = z+4 :D Programmer and the way maths and science people think are very different, but data science is marriage of huge amounts of data with computer processing which run algorithms based on scientific observation and mathematics.

Since I am lazy, explain to your self what's happening in the examples below:

Example 2:

```
z -= 3
z
```

Output:

```
13
```

Example 3:

```
z *= 17
z
```

Output:

```
221
```

Example 4:

```
z %= 5
```

Output:

```
1
```

Example 5:

```
z = 12
z ^= 2
z
```

Output:

```
144
```

Chapter 15. Strings

Video lecture for this section could be found here https://youtu.be/FXNIuZuUPLs

Get the Jupyter notebook of this blog here https://gitlab.com/datascience-book/code/-/blob/master/strings.ipynb

Imagine a perl necklace, you have a string that runs through pearls, that is you have stringed or strung the pearls. If you string bunch of characters in programming, it's called a `String`. as simple as that.

So now let's create a Hello World string as shown below:

```
"Hello World!"
```

Output:

```
"Hello World!"
```

There is a lot to learn here. We find characters `H` to `!` surrounded by double quotes `"`, they were strung. That how you create a string.

Now let's print it out using a function named `println` (I think this means print line)

```
println("Hello World!")
```

Output:

```
Hello World!
```

Now let's put a `String` into a variable named `string`

```
string =  "I am a string"
```

Output:

```
"I am a string"
```

And let's print out the variable `string`

```
println(string)
```

Output:

```
I am a string
```

Look when you print it, how neat it is without those ugly double quotes.

Now let's assign `42` to a variable named `a`

```
a = 42
```

Output:

```
42
```

Check its type. It would be `Int` anyway

```
typeof(a)
```

Output:

```
Int64
```

Now let's use a function named `repr` to convert it into `String`.

```
b = repr(a)
```

Output:

```
"42"
```

I am not sure what's the meaning or full form of `repr`, but thats a weird function name in Julia.

Take a look at the code below, execute it in your notebook.

```
c = "Answer to the ultimate question $a"
```

Output:

```
"Answer to the ultimate question 42"
```

We see that `"Answer to the ultimate question $a"`, where `$a` tells Julia to embed the value of `a` in the string. Technically this is called **String Interopolation**.

Now let's print `c` and see

```
println(c)
```

Output:

```
Answer to the ultimate question 42
```

There is a function called `occursin` that checks is a string occurs in another string. To test it, let's create a string and store it in a variable called sentence as shown below:

```
sentence = "A spammer is a person who eats spam"
```

Output:

```
"A spammer is a person who eats spam"
```

Now let's check if `"spam"` occurs in `sentence`

```
occursin("spam", sentence)
```

Output:

```
true
```

and it does!

If you want to find in which location `"spam"` occurs first, you can use function called `findfirst` as shown below:

```
findfirst("spam", sentence)
```

Ouput:

```
3:6
```

It gives the range, that is `"spam"` can be found from 3rd character to 6th character of `sentence`.

Now if you want to find all occurance of `"spam"` in `sentence`, you can use `findall` as shown below:

```
findall("spam", sentence)
```

Output:

```
2-element Vector{UnitRange{Int64}}:
 3:6
 32:35
```

Now let's check if 32nd to 35th character of senctence is really `"spam"`

```
sentence[32:35]
```

Output:

```
"spam"
```

If you have not guessed it, `String` can be thought of as a `Array` of characters, to kinda prove it, we check the 7th location of `sentence`

```
sentence[7]
```

Output:

```
'm': ASCII/Unicode U+006D (category Ll: Letter, lowercase)
```

and its `'m'`.

Now let's check what are the first seven characters of `sentence`

```
sentence[1:7]
```

Output:

```
"A spamm"
```

No let's check what we have from 7th character to the last of `sentence`, here to get the last place we use `length(sentence)`:

```
sentence[7:length(sentence)]
```

Output:

```
"mer is a person who eats spam"
```

Like we can iterate element by element in an `Array`, we can iterate character by character in a `String` and print them out as shown:

```
for character in sentence
    print("$character,")
end
```

Output:

```
A, ,s,p,a,m,m,e,r, ,i,s, ,a, ,p,e,r,s,o,n, ,w,h,o, ,e,a,t,s, ,s,p,a,m,
```

Just like we can use `join` function on `Array`, we can use them on `String` as shown:

```
join(sentence, ",")
```

Output:

```
"A, ,s,p,a,m,m,e,r, ,i,s, ,a, ,p,e,r,s,o,n, ,w,h,o, ,e,a,t,s, ,s,p,a,m"
```

This section doesn't go into regular expressions, but we can find if a regular expression is found in a `String` using the `match` function as shown:

```
m = match(r"spam", sentence)
```

Output:

```
RegexMatch("spam")
```

We might deep dive into regular expressions possibly when we are learning about Natural Language Processing.

We can use the function `uppercase` to convert all characters in a sentence to uppercase as shown:

```
uppercase(sentence)
```

Output:

```
"A SPAMMER IS A PERSON WHO EATS SPAM"
```

Similarly, there is a way to convert it to lowercase too:

```
lowercase(c)
```

Output:

```
"answer to the ultimate question 42"
```

This section is a brief introduction to `String` in Julia. I think this is enough of Strings for a Datascientist.

Chapter 16. Boolean Operations

Video lecture for this section could be found here https://youtu.be/NTxTXxbF5gY

Let's look at Boolean algebra[9] with Julia. The notebook for this blog is here <https://gitlab.com/datascience-book/code/-/blob/master/binary_arithmetic.ipynb

In Julia, something true is represented by a constant `true` and something false is represented by a constant `false`. These two things are inbuilt in Julia and you can't keep a variable like `true = 1`, Julia will throw an error.

Now let's look at operators that do binary arithmetic. The and operation is done using the and `&` operator as shown:

```
true & true
```

Output:

```
true
```

```
true & false
```

Output:

```
false
```

We use the pipe `|` symbol to do the OR operation as shown:

```
false | true
```

Output:

```
true
```

```
false | false
```

Output:

```
false
```

The tilde ~ symbol is used for a NOT as shown:

```
~ false
```

Output:

```
true
```

⊻ does the XOR operation, to type it, type \xor and press Tab key.

```
true ⊻ true
```

Output:

```
false
```

```
true ⊻ false
```

Output:

```
true
```

```
false ⊻ false
```

Output:

```
false
```

The >> is used for arithmetic right shift, so 3 divided by 2 is kinda one.

```
3 >> 1
```

Output:

```
1
```

The >>> is used for logical right shift

```
3 >>> 1
```

Output:

```
1
```

In the below example I have no clue why it became -2, may be I will explore it in later blogs

```
-3 >> 1
```

Output:

```
-2
```

I am blown away why it produces such a result in the example below, may be I will investigate when I am not lazy:

```
-3 >>> 1
```

Output:

```
9223372036854775806
```

The `<<` is used for arithmetic left shift, so shifiting `101` by `1` becomes `1010` which is 6 as shown below:

```
3 << 1
```

Output:

```
6
```

```
Now let's try out some De Morgan's
Lawfootnote:[https://en.wikipedia.org/wiki/De_Morgans_laws]:
```

```
x = true
y = false

~(x | y) == ~x & ~y
```

Output:

```
true
```

and an another one:

```
~(x & y) == ~x | ~y
```

Output:

```
true
```

[9] https://en.wikipedia.org/wiki/Boolean_algebra

Chapter 17. Comparisons

Video lecture for this section could be found here https://youtu.be/TCO6T8T8aE8

Life is full of comparisons, we must earn more next year than now. We are considered lesser than our boss because he owns an expensive car, or has more money. Programming follows real life, comparisons are important part of programming too. Let's see how its done in Julia.

The notebook for this blog is available here https://gitlab.com/datascience-book/code/-/blob/master/comparisons.ipynb. So fire your Jupyter lab and let's get started.

First let's see if 1 is greater than 2, for checking that we use greater than symbol >

```
1 > 2
```

Output:

```
false
```

Looks like 1 is less than 2 in Julia world. Now let's check if 2 is greater than 1, and it looks so as from below example:

```
2 > 1
```

Output:

```
true
```

Now let's see if 1 is greater than 1:

```
1 > 1
```

Output:

```
false
```

Looks not, and it looks as though 1 is equal to 1 as we can see below. The >= checks if the left hand side number is either greater or equal to the right hand side number.

```
1 >= 1
```

Output:

```
true
```

Now let's use the less than sign and try to do something. Let's check if 5 is less than 3 using the less than < operator

```
5 < 3
```

Output:

```
false
```

Now lets check if 3 is less than 5

```
3 < 5
```

Output:

```
true
```

Now let's see if 3 is less than or equal to 5 using the ⇐ operator. Since 3 is less than 5 it returns true.

```
3 <= 5
```

Output:

```
true
```

Now rather than putting = sign after <, we will put it before < and check.

```
3 =< 5
```

Output:

```
syntax: "<" is not a unary operator



Stacktrace:

 [1] top-level scope at In[8]:1

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

It fails.

Now let's check if 3 is less than or equal to 3.

```
3 <= 3
```

Output:

```
true
```

Now let's check of 3 equals 3, for that we use the double equal to `==` operator as shown.

```
3 == 3
```

Output:

```
true
```

Now look at the statement below,we are checking if `3 == 3.0`, that is `3` on the left side is an integer and `3.0` on the right side is a floating point number or a Real number. These two pieces of data are

stored differently in Julia in different locations, yet Julia is smart enough to compare them and say they are equal. If you want to know more about floating point and how they are stored in computers, look here https://en.wikipedia.org/wiki/Floating-point_arithmetic

```
3 == 3.0
```

Output:

```
true
```

If we want to check if two values are stored in same location in computers memory, we can use the triple equal sign `===` as shown. `3` is stored in different location than `3.0` hence Julia says they both are not equal.

```
3 === 3.0
```

Output:

```
false
```

Try this one

```
a = 5
b = 5
a === b
```

What do you infer from it?

Now there is a special value called `NaN` in Julia, it means not a number. Now it turns out that two `NaN's are not equal

```
NaN == NaN
```

Output:

```
false
```

But it turns out that in Julia all the things that are `NaN` point to a same address located in the RAM, so that passes the `===` test as shown:

```
NaN === NaN
```

Output:

```
true
```

The last operator we are going to see in the blog is not equal to, denoted by `!=`, so let's check if 4 is not equal to 5

```
4 != 5
```

Output:

```
true
```

Turns out true. Now let's check if 4 is not equal to 4:

```
4 != 4
```

Output:

```
false
```

Chapter 18. Conditions and Branching

Video lecture for this section could be found here https://youtu.be/ROvLW1-sM-w

In the last section we have seen how to use comparisons and to compare things. Now let's see how to use them. Let's see how to compare two numbers and print useful output. The notebook for this blog is here https://gitlab.com/datascience-book/code/-/blob/master/if.ipynb.

Take a look at the example below, type it in your Jupyter lab and execute it:

```
a = 5; b = 3

if a > b
    println("a = ", a, " is bigger")
end

if b > a
    println("b = ", b, " is bigger")
end

if a == b
    println("both numbers are equal")
end
```

Output:

```
a = 5 is bigger
```

It says `a = 5 is bigger`, now change the values of `a` and `b` in `a = 5; b = 3` and see what happens. Make one greater than other, make them both equal and see. Now let's see how it works. The main work horse of the above program is the `if` condition. It's syntax is as follows:

```
if <some condition>
  # Do something here if <some condition> is true
end
```

it consists of a `if` key word, followed by some condition. So in:

```
if a > b
    println("a = ", a, " is bigger")
end
```

`a > b` is the condition. If this condition is true then the code block between `if` and `end` will get executed. So in `if a > b`, `a` is 5 and `b` is 3 and hence its true, so the statement `println("a = ", a, " is bigger")` in:

```
if a > b
    println("a = ", a, " is bigger")
end
```

gets executed, you get the output `a = 5 is bigger`. Now try explaining the other parts of the program to yourself.

Now in the above program the conditions are checked three times, that is first in `if a > b` then in `if b > a` and then in `if a == b`. What if `a > b` is true, then its just waste of computing resource to check for other conditions. Is there a programming construct that when one condition passes, all others are ignored? Fortunately yes. Check the program below:

```
a = 3; b = 5

if a > b
    println("a = ", a, " is bigger")

    elseif b > a
        println("b = ", b, " is bigger")

    elseif a == b
        println("both numbers are equal")
end
```

Output:

```
b = 5 is bigger
```

In the above program `b` is greater than `a`, so when it hits `if a > b` it fails. Then Julia checks if there is a `elseif` keyword. In this case there is. First we have `elseif b > a` and `b > a` is true. So the statement under the `elseif` gets executed and hence `println("b = ", b, " is bigger")` gets executed and it prints `b = 5 is bigger`.

In the above program, we have another condition check `elseif a == b`, in reality either `a` or `b` should be greater or they must be equal, so in fact this will be hit when both conditions `a > b` and `b > a` fail, so we could avoid condition check here and assume both numbers are equal and we can write the program as shown:

```
a = 3; b = 3

if a > b
    println("a = ", a, " is bigger")

    elseif b > a
        println("b = ", b, " is bigger")

    else
        println("both numbers are equal")
end
```

Output:

```
both numbers are equal
```

Take a look at the above program. Look at the:

```
else
    println("both numbers are equal")
```

This `else` is hit only when all other conditions have failed. So when neither `a` or `b` is greater, they both must be equal. Hence there is no need of a condition check here. `else` always has no condition in it and is executed when all other conditions in `if` and `elseif` fails.

So since all conditions fail in the above program `println("both numbers are equal")` under the `else` gets execute and we get `both numbers are equal` as output.

Chapter 19. Ternary Operator

Video lecture for this section could be found here https://youtu.be/ROvLW1-sM-w

Unary operator are ones that operate on one stuff like the tilde `~` operator, you can do something like `~ a`, `~ false` etc and get a result. Binary operators operators act on two stuff, like the plus `+` operator. You can do `1 + 2` and so on, they get 2 operands. Ternary operator, operates on three stuffs. Take a look at the program below:

```
a = 10; b = 15

max = b > a ? b : a
min = b < a ? b : a

println("Maximum = ", max)
println("Minimum = ", min)
```

Output:

```
Maximum = 15
Minimum = 10
```

Type it in your Jupyter lab and execute it. So this program is able to find maximum and minimum of two values. Let's see how it works.

First we assign `a` to `10` and `b` to `15` here:

```
a = 10; b = 15
```

Next look at this line:

```
max = b > a ? b : a
```

Here we have a variable `max` and we are assigning something to it with an equal to `=` operator. The interesting part is at the right side, look at it carefully, it goes like this `b > a ? b : a`. Note the syntax here. There is a condition `b > a`, so it becomes `true` or `false` depending on the avlues of `b` and `a`, at the right of it is a `? b : a`. If `b > a` is true, the the stuff between question `?` and `:` get returned and `b` is assigned to `max`. If `b > a` is false, the stuff after the `:` is assigned to `max` that is `a`.

So this ternary operator `? :` deals with three stuffs.

1. A condition `<condition> ? :`
2. Something to be returned or done when the condition is `true <condition> ? <do something when true> :`

3. Something to be returned or done when the condition is `false` `<condition> ? <do something when true> : <do something when false>`.

I think you can figure out rest of the program by yourself.

As an exercise why don't you write a simple program with uses ternary operator and and does this:

- It multiplies two variables `a` and `b` when a variable named `action` is set to `multiply`
- It adds two variables `a` and `b` when a variable named `action` is set to any other value
- Finally it prints out the result

The Jupyer notebook for this section is available here https://gitlab.com/datascience-book/code/-/blob/master/ternary_operator.ipynb.

Chapter 20. Short Circuit Evaluation

Video lecture for this section could be found here https://youtu.be/ROvLW1-sM-w

In the section Boolean Operations, we had talked about using `&` to Boolean AND stuff and using `|` to Boolean OR. But there is one draw back. If I say `false & true`, just by looking at `false &` one can say the result is `false`, irrespective of what is on the right side operand, but this `&` looks at its both left and right side anyway. There is however are two intelligent and operators that Julia calls as short cut evaluation, one of them is `&&`, a double AND.

If there is an expression `false && true`, Julia seeing `false &&`, doesn't bother what's at the right side if the `&&` and just returns false. It saves valuable computing power. In Data Science applications we use very simple algorithms with massive amounts of data, so computing efficiency is important, so Julia is a good candidate for it. It is also possible to write very inefficient programs in Julia if one is not careful. So I feel these kind of small knowledge tidbits are useful.

So lets see a program with double AND `&&` short cut operator. Below program checks which of the three given variables have the highest values and prints them out. Note how we have used `&&` operators everywhere.

```
a = 6; b = 7; c = 7

if a > b && a > c

    println("a = ", a, " is the greatest")

elseif b > a && b > c

    println("b = ", b, " is the greatest")

elseif c > a && c > b

    println("c = ", c, " is the greatest")

else

    println("All or some variables have equal value")
end
```

Output:

```
All or some variables have equal value
```

So if you take the line `if a > b && a > c`, when `a > b` is false, Julia does not compare `a` with `c` in ` a > c`. That's bit smart and if this program is running billion times on say 50 data crunching servers, it makes a difference.

Similarly we have a double pipe symbol `||` used for OR operations. So if there is `true || false` situation, Julia never bothers what's at the right side of `||`, it just returns `true`. Below is a program that takes weather input, if the weather is `"sunny"` or `"rainy"`, it tells you to take an umbrella, else it says its a nice weather.

```
weather = "sunny"

if (weather == "sunny") || (weather == "rainy")
    println("Take an umbrella")
else
    println("Looks like nice weather")
end
```

Output:

```
Take an umbrella
```

The notebook for this blog is available here https://gitlab.com/datascience-book/code/-/blob/master/short_circuit_evaluation.ipynb.

20.1. How not to use Shortcut Evaluations

In my Jupyter notebook on Shortcut Evaluations here https://gitlab.com/datascience-book/code/-/blob/master/short_circuit_evaluation.ipynb., I had tabled such an example for you to fiddle:

```
y = false # change this to true and false and see what happens

y && println("Right side executed")
```

This might make you think to write programs like this:

```
print_stuff = true

print_stuff && println("Stuff") # very bad way of programming
```

Where `&&` in the above example is used as some kind of condition branching like `if`. This might seem appealing and really short and concise way of expressing thing, that is print something or so some operation like this:

```
a = 5
increment = false

increment && (a += 1) # remove the paranthesis and see what happens, can you explain
why?
increment || (a -= 1) # remove the paranthesis and see what happens, can you explain
why?

println("a = ", a)
```

In the above example a gets incremented if `increment` is `true`, and decremented if `increment` is `false`. Well in programming terms this is bullshit. Absolute bullshit. Constructing these types of programs may not invite a frown from the Julia compiler, but will definitely invite frowns from a good programmer.

`&&` and `||` are not supposed to be used as conditional branching. They are used for logical operations. So as a responsible programmer / data scientist (who's job is programming too), it's your job to create readable, understandable and maintainable programs. So possibly you could write the above program as shown:

```
# The right way

a = 5
increment = false

if increment
    a += 1
else
    a -= 1
end

println("a = ", a)
```

or like this:

```
# Right and bit concise way

a = 5
increment = true

increment ? (a += 1) : (a -= 1)

println("a = ", a)
```

Personally I am against the concise way because it does not make the program good to read. It's my opinion, using the `if` and spreading the things over multiple lines makes a clear read I would say.

The Jupyter notebook for this blog is available here https://gitlab.com/datascience-book/code/-/blob/master/how_not_to_use_shortcut_evaluations.ipynb.

Chapter 21. While Loops

Video lecture for this section could be found here https://youtu.be/oNCX8MWxI7E

No matter what modern programming language you take, you will find a while loop in it, Julia is no exception in that case (but there are languages without it). This section is about while loops in Julia. The notebook for this section can be found here: https://gitlab.com/datascience-book/code/-/blob/master/while.ipynb. So let's try it out and see. Take this first example, type it in your Jupyter lab and execute it:

```
i = 1

while i <= 10
    println(i)
    i += 1
end
```

Output:

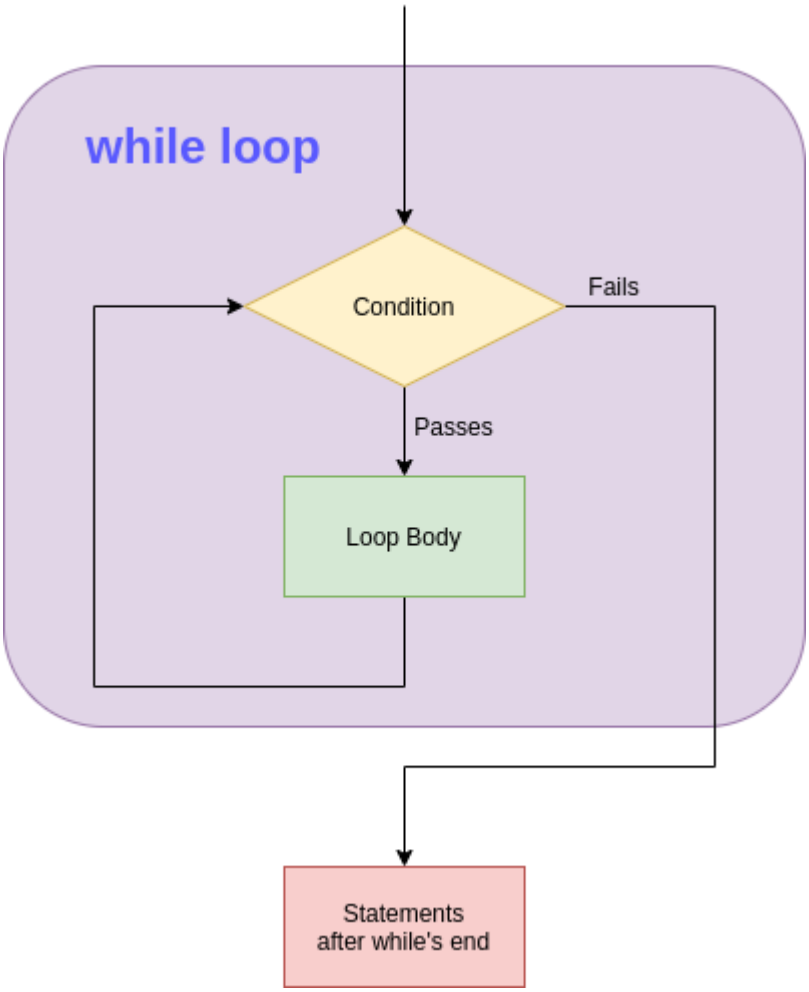
```
1
2
3
4
5
6
7
8
9
10
```

So how it works? First we have got this statement `i = 1`, where we initialize a variable named `i` to `1`. Next we have `while i ⇐ 10`, so this `while` key word gets a condition `i ⇐ 10`, and `i` is less than 10, so the stuff between `while <condition>` and `end` gets executed. So in the the above code they are these lines:

```
println(i)
i += 1
```

Here we print `i` in `println(i)` and next we increment `i` by `1` here `i += 1` and hence `i` becomes `2`. When it encounters `end`, it does not end, the control transfers back to `while i ⇐ 10`, since the condition is true again the loop body gets executed again and `2` gets printed and `i` become `3` now. It goes on till `i` is `11` and when it hits `while i ⇐ 10`, `i ⇐ 10` fails and the loop never gets executed and the program ends.

For a visual look here is a diagram that explains how a while loop works:



A while or most programming loops needs the three things to work properly

1. A variable initiation, for us its `i = 1`.
2. A condition check, for us its `i <= 10`.
3. Variable update, for us its `i += 1`.

If we leave out (3), the loop could be a infinite loop, comment out `i += 1` and see what happens, hope you know to stop your kernel and kill Jupyter notebook / lab, refer Installing Jupyter notebook and Jupyter lab.

21.1. Finding primes

So let's now see an example where we can put our `while` loop to use, type the program below and execute it. Change the value of `number` and see. This program finds if a number is prime or not.

```
number = 71

counter = 2

while number % counter != 0
    counter += 1
end

if number != counter
    print(number, " is not prime")
else
    print(number, " is prime")
end
```

Output:

```
71 is prime
```

Let's see how it works. If you know math, then you would know that primes are divisible only by 1 and it self. So if any number is prime then it gives a non zero reminder if it divided by any other number other than itself or 1. In the above program that's what we do. We have a variable `counter` and we set it to `2` here `counter = 2`, next we have this code block:

```
while number % counter != 0
    counter += 1
end
```

now look at the condition, if the number is prime and not 2, this condition `number % counter != 0` will be true, and if you are wondering `!=` stands for, it stands for not equal to in programming. So if the condition is true, the code between `while` and `end` executes, hence counter now becomes 3 by executing `counter += 1`, and once again `71 % 3` is not zero and so again `counter` increments and in next iteration it is `71 % 4` and so on. The loop ends when `counter` is `71`, where `71 % 71` is `0` and hence `number % counter != 0` fails and the program control comes out of the `while`.

It (the program execution) now encounters this:

```
if number != counter
    print(number, " is not prime")
else
    print(number, " is prime")
end
```

Here in the above piece of code if the `number` is not equal to `counter` we print it's not a prime, but when it is, we print is as prime.

The notebook for this blog can be found here: https://gitlab.com/datascience-book/code/-/blob/

master/while.ipynb.

Chapter 22. Ranges and for loops

Video lecture for this section could be found here https://youtu.be/3cuTFzcW8KQ

If you wand to say 1 to 10 in Julia, this is a simpler way to do it, its called as range:

```
1:10
```

Output:

```
1:10
```

It's syntax is `<start value>:<end value>`. And you can print 1 to 10 like this:

```
for i in 1:10
    print(i, ' ')
end
```

Output:

```
1 2 3 4 5 6 7 8 9 10
```

Let me introduce `for` loop, this can remove lot of clutter compared to a `while` loop. This `for` works like this, first you have a variable in this case `i` and then it's followed by a `in` keyword and is followed by a range (in future we will see other data types as well). So you can think `for` as something that un-bundles `1:10` and puts each value into the variable `i` one at a time and executes the loop body that is in the above case is `print(i, ' ')`. Notice how the `print()` is different from the `println()`. `println()` prints a new line at the end whereas `print()` does not.

Let's say we want to print numbers from 5 to 100 in steps of 5, range has a way for that too, take a look at the example below:

```
for i in 5:5:100
    print(i, ' ')
end
```

Output:

```
5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
```

The format for range with step is this `<start value>:<step size>:<end value>` as shown below:

```
5:5:100
```

Output:

```
5:5:100
```

The notebook for this blog is here https://gitlab.com/datascience-book/code/-/blob/master/ranges.ipynb.

Chapter 23. Breaks and Continues

Sometimes while you are executing a loop, you might like to skip something, at those times you can use a key word called `continue`. Let's say that I don't want 6 getting printed when I write a program to print 1 to 10, I can write a program like this:

```
i = 0

while i < 10

    i += 1

    if i == 6
        continue
    end

    println(i)
end
```

Output:

```
1
2
3
4
5
7
8
9
10
```

Look at these lines of code in the above example:

```
if i == 6
    continue
end
```

It tells Julia to continue to next iteration if `i` equals `6`, so the statement that comes after it, that is in this case `println(i)` won't get executes when `i` is `6`, and it will go to the next iteration and `i` will become `7` and the loop will go on normally. So seeing this why can't you write a program that prints only even numbers or odd numbers below a given value in the number line?

`continue` tells Julia to skip operation and go on to next iteration, `break` tells it to break out of the loop entirely. Type the program below in your notebook and execute it:

```
i = 1

while i <= 10

    println(i)

    if i == 6
        break
    end

    i += 1
end
```

Output:

```
1
2
3
4
5
6
```

Here when `i` is `6`, in these lines:

```
if i == 6
    break
end
```

when `i==6` becomes true, and `break` will get executed, and the program breaks out of the loop. Hence only 1 to 6 gets printed and the loop breaks.

A very similar implementation of the above programs is given below using `for` loop for `continue`:

```
for i in 1:10

    if i == 6
        continue
    end

    println(i)
end
```

Output:

```
1
2
3
4
5
7
8
9
10
```

and this one is for `break`:

```
for i in 1:10
    println(i)

    if i == 6
        break
    end
end
```

Output:

```
1
2
3
4
5
6
```

The notebook for this blog can be found here https://gitlab.com/datascience-book/code/-/blob/master/breaks_and_continues.ipynb.

Chapter 24. Arrays

Video lecture for this section could be found here https://youtu.be/qCPIr-vD1Ls

Arrays are nothing but a collection of some stuff. Imagine a rack with, where each compartment is numbered/indexed 1, 2 ,3 ... and so on and you put an item in each compartment. At a later time you can retrieve the item using the index.

The code for this blog can be fount here https://gitlab.com/datascience-book/code/-/blob/master/arrays.ipynb, let's get practical here now. Launch your Jupyter Lab and try these out:

First lets create an array as shown below:

```
array = [1, 2, 3, 4, 5]
```

Output:

```
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

As you can see we have `1, 2, 3, 4, 5`, within `[` and `]`, that's how Julia knows we have created an array, and we assign it to variable named `array`. So the variable `array` points to the start of `[1, 2, 3, 4, 5]`.

To determine how many elements are present in an array, we use the `length()` function as shown below

```
length(array)
```

Output:

```
5
```

I don't think it should be a mystery to you that we have 5 elements in `array`.

Now let's try `size()`:

```
size(array)
```

Output:

```
(5,)
```

so it returns something weird, and it looks like this `(5,)`, this is called a Tuple, a very similar data structure like Array, but it's immutable. That is we cannot change it. We will look at it in the later blogs.

Now let's get into some kind of statistics, what about totaling the values in an array, take a look at the example below:

```
total = 0

for element in array
    total += element
end

println(total)
```

Output:

```
15
```

We use the `for` loop here, in this line `for element in array`, in each iteration, `element` get the value of one element of `array`. That is in the first iteration `element` becomes `1` in the second `2` and so on. Now when it hits `total += 0`, in the first iteration total becomes `0 + 1`, that is `1`, in the second iteration `1 + 2` that is `3` and so on.... Finally we print `total` here `println(total)` and its `15`.

Now let's see what the average is. Its nothing but `total` divided by number of elements in the array as shown below:

```
avg = total / length(array)
```

Output:

```
3.0
```

In previous blogs we have seen how to find maximum of 2 or three numbers, but look at the program below. You can pack how many number you want in the `array` and it will find the maximum:

```
max = array[1]

for element in array
    max = max > element ? max : element
end

println(max)
```

Output:

```
5
```

First we have a variable `max` which s assigned to the first element of the `array` here `max = array[1]`, now we have these lines:

```
for element in array
    max = max > element ? max : element
end
```

In the above lines, each element of the `array` is assigned to `element` and here `max = max > element ? max : element`, `max` takes on the value of `element` if it's greater than `element` and finally we print out `max` here `println(max)`.

Now let's see how to access 3rd element of an array:

```
array[3]
```

Output:

```
3
```

Now we access elements 2 to 4:

```
array[2:4]
```

Output:

```
3-element Array{Int64,1}:
 2
 3
 4
```

See that in the above example we pass range `2:4` between the brackets.

We wrote a long program to find maximum of an array, but we can also do it like this:

```
maximum(array)
```

Output:

```
5
```

Here is the code to find minimum:

```
minimum(array)
```

Output:

```
1
```

We can get cumulative sum of an array as shown:

```
cumsum(array)
```

Output:

```
5-element Array{Int64,1}:
  1
  3
  6
 10
 15
```

This is how to find total or sum of an array:

```
sum(array)
```

Output:

```
15
```

Now let's multiply all elements of an array by 7:

```
array * 7
```

Output:

```
5-element Array{Int64,1}:
  7
 14
 21
 28
 35
```

Now let's add 1 to all elements of the array:

```
array.+ 1
```

Output:

```
5-element Array{Int64,1}:
 2
 3
 4
 5
 6
```

Note that we use a dot in `array.+ 1`, the dot `.` is a broadcasting operator which is used for element wise operations. I find that we can omit `.` for multiplying and dividing array by scalar.

In the example below we divide array by scalar:

```
array / 4
```

Output:

```
5-element Array{Float64,1}:
 0.25
 0.5
 0.75
 1.0
 1.25
```

In the below example we subtract 4 from each element of the array, notice that we use the dot `.` operator:

```
array.- 4
```

Output:

```
5-element Array{Int64,1}:
 -3
 -2
 -1
  0
  1
```

There is a function called `typeof()` in Julia that tells the variable type. Let's see what data type `array` is:

```
typeof(array)
```

Output:

```
Array{Int64,1}
```

So it says `Array`, that is collection of stuff in an ordered format, it's elements are integers `Int`, and each element occupies `64` bits of space. Why don't you check what is the type of `array / 4`?

Now let's see how to add a new element to an array:

```
push!(array, 21)
```

Output:

```
6-element Array{Int64,1}:
  1
  2
  3
  4
  5
 21
```

In the above case we have added `21` to the array using `push!()` function. As a first argument to `push!()` we give `array` and as second argument we give the element to be added that is `21`.

Wen ever you use `push!()`, you see that array gets modified as shown below. The exclamation mark, also called as bang `!` in computer science indicates that the argument been passed to it gets modified. Let's check the value of `array` now:

```
array
```

Output:

```
6-element Array{Int64,1}:
  1
  2
  3
  4
  5
 21
```

Now let's pop out the last element of the array:

```
pop!(array)
```

Output:

```
21
```

We use function called `pop!()`, since it comes with a bang `!`, we can expect `array` is modified here as shown below:

```
array
```

Output:

```
5-element Array{Int64,1}:
 1
 2
 3
 4
 5
```

We push `3` into the array:

```
push!(array, 3)
sort(array)
```

Output:

```
6-element Array{Int64,1}:
 1
 2
 3
 3
 4
 5
```

We check it again as though we don't believe in `push!()`:

```
array
```

Output:

```
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 3
```

We reverse the order of elements in array

```
reverse(array)
```

Output:

```
6-element Array{Int64,1}:
 3
 5
 4
 3
 2
 1
```

We sort the array here:

```
sort!(array)
```

Output:

```
6-element Array{Int64,1}:
 1
 2
 3
 3
 4
 5
```

Since we used a function with bang `!` above, we can expect the result to be stored in variable `array` as shown below:

```
array
```

Output:

```
6-element Array{Int64,1}:
 1
 2
 3
 3
 4
 5
```

Now let's do some real math, lets say we are buying some items, their quantities and prices are given below:

```
mangoes = 2
rice_in_kg = 5
eggs = 12

mango_price = 50
rice_price = 30
egg_price = 7

quantities = [mangoes, rice_in_kg, eggs]
prices = [mango_price, rice_price, egg_price];
```

The above cell does not spit any output, look at the semicolon `;` at the end of the last line that prevents the cell from cluttering your notebook with its outputs.

Since it's Julia, I can compute the total prices as shown below:

```
total_array = quantities.* prices
```

Output:

```
3-element Array{Int64,1}:
 100
 150
  84
```

As you can see, you can compute the total cost of each item using just the star * operator. That's a lot of savings for you from writing lot of iterative code (if you know to code in other languages).

Now let's compute the sun of the above `total_array`

```
grand_total = sum(total_array)
```

Output:

```
334
```

Julia provides `sum()` function that computes the total of an array.

Now let's use Julia's built in linear algebra package, and since `quantities` and `prices` are nothing but vectors, we can compute the total by taking it's dot product as shown:

```
using LinearAlgebra
dot(quantities, prices)
```

Output:

```
334
```

In the above example we have learned to use or import a package, to import it we use `using <package name>`. Thankfully Julia has `LinearAlgebra` package built into it, but it's not imported at the start by default, so we need to import it using `using LinearAlgebra`. It provides a convenient `dot()` function to calculate the dot product.

Now say we want to be more concise, we can use the dot operator provided by `LinearAlgebra` package.

```
quantities  prices
```

Output:

```
334
```

In case you are wondering how to type ⋅, type `\cdot` and press `Tab`.

Chapter 25. Tuples

Video lecture for this section could be found here https://youtu.be/Uahv2THnDfY

Tuple is a data type like Array, the only thing is you can't change values in a Tuple, in Array you can. You can get the Jupyter notebook for this blog here https://gitlab.com/datascience-book/code/-/blob/master/tuples.ipynb.

So let's dive in.

First we create a Tuple and assign it to a variable called `tuple`

```
tuple = (1, 5, 9, 3, 7, 2)
```

Output:

```
(1, 5, 9, 3, 7, 2)
```

So the above example speaks something, in Arrays we wrap elements with square brackets `[]`, but here we are wrapping it with rounded ones `()`.

In the below example we check the Tuple's type:

```
typeof(tuple)
```

Output:

```
NTuple{6,Int64}
```

I am not sure why its called `NTuple`, I read some docs and I infer it means a homogeneous data type. In the above example our `tuple` contains all `Int64` that is integers of 64 bit wide; and it has `6` elements in them, identified by `6` in `NTuple{6,Int64}`.

An array is mutable, that is the values in it can be changed, below we have an Array:

```
array = [1, 5, 9, 3, 7, 2]
```

Output:

```
6-element Array{Int64,1}:
 1
 5
 9
 3
 7
 2
```

and we change the third element in it:

```
array[3] = 10
array
```

Output:

```
6-element Array{Int64,1}:
  1
  5
 10
  3
  7
  2
```

This statement `array[3] = 10` accesses the third element and sets it to `10`. If we try to do this in Tuple as shown below, we get an error:

```
tuple[3] = 10
tuple
```

Output:

```
MethodError: no method matching setindex!(::NTuple{6,Int64}, ::Int64, ::Int64)



Stacktrace:

 [1] top-level scope at In[5]:1

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

Like an Array we can get maximum value in a Tuple using the `maximum()` function:

```
maximum(tuple)
```

Output:

```
9
```

We can get minimum value in a Tuple using the `minimum()` function:

```
minimum(tuple)
```

Output:

```
1
```

We can get total of Tuple using the `sum()` function:

```
sum(tuple)
```

Output:

```
27
```

Below we get the cumulative sum using `cumsum()` function:

```
cumsum(tuple)
```

Output:

```
(1, 6, 15, 18, 25, 27)
```

We are unable to sort it however, I did not try out `sort!()` because anything with a bang modifies the argument that's been passed to it. A Tuple is immutable and hence `sort!()` makes no sense and hence I tried `sort()` and it did not work:

```
sort(tuple)
```

Output:

```
MethodError: no method matching sort(::NTuple{6,Int64})
Closest candidates are:
  sort(!Matched::AbstractUnitRange) at range.jl:1014
  sort(!Matched::AbstractRange) at range.jl:1017
  sort(!Matched::SparseArrays.SparseVector{Tv,Ti}; kws...) where {Tv, Ti} at
/buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.5/SparseArrays/src
/sparsevector.jl:1912
  ...



Stacktrace:

 [1] top-level scope at In[14]:1

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

Possibly one must convert Tuple into Array to sort.

Now let's use our `LinearAlgebra` package to and wee if operator works on Tuple:

```
using LinearAlgebra
tuple  tuple
```

Output:

```
169
```

and it does. To get the dot operator type `\cdot` followed by `Tab`.

In our last blog about [Array]({% post_url 2020-11-17-arrays %}), we saw that when getting size of an array we got something as shown:

```
size_of_array = size(array)
```

Output:

```
(6,)
```

Well the size of an array is a Tuple. You also see that in the above example it gives an Output `(6,)` and not `(6)`. Now let's inspect the type of the returned value:

```
typeof(size_of_array)
```

Output:

```
Tuple{Int64}
```

As an exercise, why don't you try `typeof6` and see what it is? A Tuple should always have an comma `,`, thats how Julia identifies it's an Tuple than a number surrounded by brackets.

I wish we could convert Tuple to Array by using a construct like this `Tuple(<array>)`, but it turns out that you must do a trickery as shown below:

```
array_from_tuple = [element for element in tuple]
```

Output:

```
6-element Array{Int64,1}:
 1
 5
 9
 3
 7
 2
```

In the above example notice this `[element for element in tuple]`, this is called list comprehension, it works like this, it starts with an empty Array `[]`, now take this `for element in tuple`, this is a `for` loop, so for each element in `tuple`, it take an element one by one and puts it into variable named `element`, now this element is added to Array here `[element ⋯.]`, so in the first iteration it becomes `[1]`, in the next `[1, 5]` and so on... Finally we get an Array `[1, 5, 9, 3, 7, 2]` which gets assigned to `array_from_tuple`.

Though making an Array from Tuple is a pain, making a Tuple from an Array seems to be as easy as shown:

```
tuple_from_array = Tuple(array)
```

Output:

```
(1, 5, 10, 3, 7, 2)
```

weird Julia.

Chapter 26. Comprehension

Video lecture for this section could be found here https://youtu.be/MmVaj9qIjX8

In this blog let's look at comprehensions, you can get the notebook here https://gitlab.com/datascience-book/code/-/blob/master/comprehension.ipynb. First we define a range as shown:

```
range = 1:10
```

Output:

```
1:10
```

Now say we want to square the values in the range, we can do it as shown:

```
[value^2 for value in range]
```

Output:

```
10-element Array{Int64,1}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

Look at the construct of the program, we separate out each item in the `range` using this statement `for value in range`, each time the item gets stored in a variable called `value`, and in `value^2` we square it, and we capture it in an array by wrapping it all between square brackets like this `[value^2 for value in range]`. This trickery is called list comprehension.

Below let me introduce to you the `rand()` function, in the below example we want to generate random number between 1 and 100, so we use `rand(1:100)`, and we want to generate 10 numbers, so we pass `10` as the second argument as ten like this `rand(1:100, 10)`. Type the example below and execute:

```
array = rand(1:100, 10)
```

Output:

```
10-element Array{Int64,1}:
  70
  41
  87
  48
  54
  63
  61
 100
  26
  37
```

As you see from the above example, we assign the random numbers to a variable called `array`. Now let's use list comprehension to take square root of array as shown:

```
square_root = [value^0.5 for value in array]
```

Output:

```
10-element Array{Float64,1}:
  8.366600265340756
  6.4031242374328485
  9.327379053088816
  6.928203230275509
  7.3484692283495345
  7.937253933193772
  7.810249675906654
 10.0
  5.0990195135927845
  6.082762530298219
```

Okay, list comprehension is very powerful feature, but one can take square of array as shown:

```
array.* array
```

Output:

```
10-element Array{Int64,1}:
  4900
  1681
  7569
  2304
  2916
  3969
  3721
 10000
   676
  1369
```

So in the above example we do element wise multiplication, for that we use the `.*` operator. I tried out `array * array` and hoped that it would work, but it did not.

The dot `.` means element wise operation. We can do element wise squaring of the `array` as shown:

```
array.^2
```

Output:

```
10-element Array{Int64,1}:
  4900
  1681
  7569
  2304
  2916
  3969
  3721
 10000
   676
  1369
```

All we do add a dot `.` before `^` and that's it. So does it mean list comprehension is dead? Nope we can do some trickery with list comprehension as shown:

```
odd_numbers = [value for value in array if value % 2 != 0]
```

Output:

```
5-element Array{Int64,1}:
 41
 87
 63
 61
 37
```

Let's see how the program works. First we take each item in the `array` and assign it to a variable called `value` here `for value in array`, but we do not do that all the time, it happens only when the value is odd as given by `if value % 2 != 0`, if that happens we take the `value` and put in in an Array like this `[value ⋯..]`. So we have complete program as shown here `[value for value in array if value % 2 != 0]`. So what we are doing is filtering the odd numbers, nothing else.

26.1. Generator Comprehension

Just like list comprehension, in which things are wrapped around by square braces, we use round braces for a thing called generator comprehension. If you look at the example below, we do squaring elements of `range`:

```
squares = (el^2 for el in range)
```

Output:

```
Base.Generator{typeof(range),var"#3#4"}(var"#3#4"(), range)
```

but we do it inside round braces, so we get an output as shown above. That is squares are not evaluated unless absolutely necessary. So to force the evaluation we run the code below

```
join(squares, ", ")
```

Output:

```
"1, 4, 9, 16, 25, 36, 49, 64, 81, 100"
```

Where the method `join()`, joins the element of passed collection `squares` in this case with a string `", "`.

26.2. Permutation

If we roll two dice, let's see what permutations we get. Type the code below and execute:

```
dice_range = 1:6

# Finding permuation for roll of two dice

dice_permuations = [(x, y) for x = dice_range, y = dice_range]
```

Output:

```
6×6 Array{Tuple{Int64,Int64},2}:
 (1, 1)  (1, 2)  (1, 3)  (1, 4)  (1, 5)  (1, 6)
 (2, 1)  (2, 2)  (2, 3)  (2, 4)  (2, 5)  (2, 6)
 (3, 1)  (3, 2)  (3, 3)  (3, 4)  (3, 5)  (3, 6)
 (4, 1)  (4, 2)  (4, 3)  (4, 4)  (4, 5)  (4, 6)
 (5, 1)  (5, 2)  (5, 3)  (5, 4)  (5, 5)  (5, 6)
 (6, 1)  (6, 2)  (6, 3)  (6, 4)  (6, 5)  (6, 6)
```

So let's see how the code works. So we are using list comprehension, and we are wrapping the output in square brackets `[]`, we are wrapping a Tuple inside a list `[(x, y)]`, and the Tuple is fed by variables `x` and `y`. These `x` and `y` take their values in an iterative manner from `dice_range` that ranges from 1 to 6 here `for x = dice_range, y = dice_range`. So this concise piece of code generates us a 6X6 array with all permutations of `x`and `y`. Below we check the size of the permutation as though we don't trust Julia:

```
size(dice_permuations)
```

Output:

```
(6, 6)
```

26.3. Flattened Comprehension

The comprehension discussed above returned a rank 2 matrix, but what if want to generate a flattened one? See the example below:

```
flattened_permuations = [(x, y) for x = dice_range for y = dice_range]
```

Output:

```
36-element Array{Tuple{Int64,Int64},1}:
 (1, 1)
 (1, 2)
 (1, 3)
 (1, 4)
 (1, 5)
 (1, 6)
 (2, 1)
 (2, 2)
 (2, 3)
 (2, 4)
 (2, 5)
 (2, 6)
 (3, 1)
 □
 (5, 1)
 (5, 2)
 (5, 3)
 (5, 4)
 (5, 5)
 (5, 6)
 (6, 1)
 (6, 2)
 (6, 3)
 (6, 4)
 (6, 5)
 (6, 6)
```

In the above example we simply use 2 `for`□`s like this` `` `for x = dice_range for y = dice_range ``. This generates us a vector of Tuple's rather than a matrix as in the previous example. Now let's check its size to conform it's a vector:

```
size(flattened_permuations)
```

Output:

```
(36,)
```

Chapter 27. Sets

Video lecture for this section could be found here https://youtu.be/ju7J3pBU7fM

Julia has a datatype called `Set`, that is mathematical equivalent of a [Set](https://en.wikipedia.org/wiki/Set_(mathematics)). From the angle of Julia you can think `Set` as an `Array` which contains just unique values and cannot be ordered or sorted.

So let's dive in, we can create a set as shown below:

```
set = Set([1, 2, 3, 4])
```

Output:

```
Set{Int64} with 4 elements:
  4
  2
  3
  1
```

So we see that we use an array `[1, 2, 3, 4]`, this is been passed to a Function called `Set()` like this `Set([1, 2, 3, 4])` and out we get out a set which is stored in a variable called `set`. Why don't you try out `Set([1, 2, 3, 4, 1, 2, 3, 4])` and see what happens?

We will also create another set as shown:

```
another_set = Set((3, 4, 5, 6, 7))
```

Output:

```
Set{Int64} with 5 elements:
  7
  4
  3
  5
  6
```

Now we will try to push in a new value `8` in another set. Note we use `push!()` function which means the argument passed to it gets modified:

```
push!(another_set, 8)
```

Output:

```
Set{Int64} with 6 elements:
  7
  4
  3
  5
  8
  6
```

In the above piece of code we give `another_set` as argument to the `push!()` and we add 8 to it, so `another_set` get modified.

Let's now try to push `8` again to `another_set`:

```
push!(another_set, 8)
```

Output:

```
Set{Int64} with 6 elements:
  7
  4
  3
  5
  8
  6
```

As you can see from the above example, a set contains just unique values and if you push a value that already exists, it will not grow.

27.1. Unions

Julia has a `union()` function with which you could get a set's union as shown:

```
union(set, another_set)
```

Output:

```
Set{Int64} with 8 elements:
  7
  4
  2
  3
  5
  8
  6
  1
```

27.2. Intersection

Julia has a `intersect()` function with which you can find intersection of two sets as shown:

```
intersect(set, another_set)
```

Output:

```
Set{Int64} with 2 elements:
  4
  3
```

27.3. Difference

Julia has `setdiff()` function with which you can find the difference of two sets as shown, in the belowcode it means get all the elements of `set` that are not there in `another_set`:

```
setdiff(set, another_set)
```

Output:

```
Set{Int64} with 2 elements:
  2
  1
```

27.4. Other Operations

You can find number of elements in a set using `length()` function as shown:

```
length(set)
```

Output:

```
4
```

To get the type of set you can use the `typeof()` operator as shown:

```
typeof(set)
```

Output:

```
Set{Int64}
```

Set is just another collection so you can use `for` loop as shown:

```
for element in set
    print(element, ", ")
end
```

Output:

```
4, 2, 3, 1,
```

Rather than using the cumbersome `for`, you can print the values of set as shown:

```
print(join(set, ", "))
```

Output:

```
4, 2, 3, 1
```

In the above example `join()`, joins the `set` elements with a string `", "` and outputs this string: `"4, 2, 3, 1"`, this when passed to `println()` outputs `4, 2, 3, 1`.

27.5. You can't sort a Set

You can't sort a set, unlike `Array` and `Tuple` a set is unordered:

```
sort(set)
```

Output:

```
MethodError: no method matching sort(::Set{Int64})
Closest candidates are:
  sort(!Matched::AbstractUnitRange) at range.jl:1014
  sort(!Matched::AbstractRange) at range.jl:1017
  sort(!Matched::SparseArrays.SparseVector{Tv,Ti}; kws...) where {Tv, Ti} at
/buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.5/SparseArrays/src
/sparsevector.jl:1912
  ...



Stacktrace:

 [1] top-level scope at In[12]:1

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

27.6. Converting Set to Array

You can't convert a set to array as easy as passing a set to an `Array()` as shown:

```
Array(set)
```

Output:

```
MethodError: no method matching Array(::Set{Int64})
Closest candidates are:
  Array(!Matched::LinearAlgebra.SymTridiagonal) at
/buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/sr
c/tridiag.jl:142
  Array(!Matched::LinearAlgebra.Tridiagonal) at
/buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/sr
c/tridiag.jl:583
  Array(!Matched::LinearAlgebra.AbstractTriangular) at
/buildworker/worker/package_linux64/build/usr/share/julia/stdlib/v1.5/LinearAlgebra/sr
c/triangular.jl:157
  ...

Stacktrace:

 [1] top-level scope at In[13]:1

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

You need to use list [comprehension]({% post_url 2020-11-20-comprehension %}) trickery as shown:

```
array_from_set = [element for element in set]
```

Output:

```
4-element Array{Int64,1}:
 4
 2
 3
 1
```

As though we do not believe we can make `Array` from `Set` we check the typeof of `array_from_set`

```
typeof(array_from_set)
```

Output:

```
Array{Int64,1}
```

27.7. Coverting Set to Tuple

We do not get an `Array` from set using the `Array()` thing, but we can get a `Tuple`, using `Tuple()` as shown:

```
Tuple(set)
```

Output:

```
(4, 2, 3, 1)
```

That's sad, because `Array` is treated as unwanted second class citizens, but may be there could be a reason for that which I am unaware.

27.8. Pop out a element from a Set

We can pop out a element from a set using the `pop!()` function as shown:

```
pop!(set)
```

Output:

```
4
```

Note that since we have used bang `!`, it modifies the passed variable `set` and the popped out value `4` get removed from `set` as shown below:

```
set
```

Output:

```
Set{Int64} with 3 elements:
  2
  3
  1
```

`size()` returns a `Tuple` and its for multidimensional data structure like `Array`, so it won't work on `Set`:

```
size(set)
```

Output:

```
MethodError: no method matching size(::Set{Int64})
Closest candidates are:
  size(!Matched::BitArray{1}) at bitarray.jl:104
  size(!Matched::BitArray{1}, !Matched::Integer) at bitarray.jl:107
  size(!Matched::Core.Compiler.StmtRange) at show.jl:1874
  ...



Stacktrace:

 [1] top-level scope at In[19]:1

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

Well that's it for sets, get the Jpyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/sets.ipynb.

Chapter 28. Dictionaries

Video lecture for this section could be found here https://youtu.be/1U2XSGMSJWg

Jupyter notebook for this section could be found here https://gitlab.com/datascience-book/code/-/blob/master/dictionary.ipynb

What comes to your mind when you think of a dictionary? Today in software or online dictionary you type a word and get its meaning in seconds. When I was small we had paper books, in each page there were printed lot of words and its meaning, even then we could find meaning of a word within one or two minutes. That's because dictionary was a ordered set of words with which we could pin point the word were looking with ease. In programming languages dictionaries are designed in similar fashion, it could book looked up with ease.

In a language dictionary, you can think of a looked up word as a key or index, the meaning that is entered against the word as a value. Take a look at the code below, type it and execute it:

```
spell = Dict(
    1 => "one",
    2 => "two",
    3 => "three"
)
```

Output:

```
Dict{Int64,String} with 3 entries:
  2 => "two"
  3 => "three"
  1 => "one"
```

In the above example we create a dictionary using the `Dict()` function, the keys are `1`, `2` and `3`, we have given their English names as their values. Now if you want to get the spelling of `3`, you can get it as shown:

```
number = 3
spell[number]
```

Output:

```
"three"
```

One who knows arrays can argue this can be achieved with the following code:

```
spell = ["one", "two", "tree"]
spell[3]
```

that's correct, but take the example as shown below:

```
prices = Dict(
    "mango" => 50,
    "bananna" => 10,
    "samosa" => 15,
    "briyani" => 80
)
```

Output:

```
Dict{String,Int64} with 4 entries:
  "mango"   => 50
  "samosa"  => 15
  "briyani" => 80
  "bananna" => 10
```

above we have created a dictionary with prices of items, now say is it better for one to access price of a mango by giving `prices[1]`, or `prices["mango"]`?

Let's say we went to shop and bought two Briyani parcels and four Samosas, I'm starting to feel hungry now . We can represent our purchase as shown:

```
purchase = Dict(
    "briyani" => 2,
    "samosa" => 4
)
```

Output:

```
Dict{String,Int64} with 2 entries:
  "samosa"  => 4
  "briyani" => 2
```

Just to know more about dictionaries, this is how you would list keys of a dictionary:

```
keys(purchase)
```

Output:

```
Base.KeySet for a Dict{String,Int64} with 2 entries. Keys:
  "samosa"
  "briyani"
```

This is how you will list values contained in the dictionary:

```
values(purchase)
```

Output:

```
Base.ValueIterator for a Dict{String,Int64} with 2 entries. Values:
  4
  2
```

The `length()` function works in a dictionary and returns the count of key value pairs in it.

```
length(purchase)
```

Output:

```
2
```

`typeof()` seems to be a universal function, and it works on dictionaries as well:

```
typeof(purchase)
```

Output:

```
Dict{String,Int64}
```

Now let's calculate our bill, type the program below and see what happens:

```
total = 0

for (item, quantity) in purchase
    total += prices[item] * quantity
end

total
```

Output:

```
220
```

So let me explain how the program works. First we have assigned a variable `total` to `0` here `total = 0`.

Next look at this statement:

```
for (item, quantity) in purchase
```

look how we unravel the key value pair as `(item, quantity)`. Since we are using a `for` loop, for every purchase, the key gets stored in variable called `item` and value get's stored in variabe called `quantity`. Now for the first iteration it would would be like this:

`total += prices[item] * quantity`

Which would translate into `total = 0 + (prices["briyani"] * 2)`, which will be reduced to `total = 80 * 2`, hence total will become `160`, and the loop goes on and on till all the `purchase` elements are fetched and the `total` calculated, and we get a grand total of `220`

Since we know list comprehension, we can write the above code as shown below:

```
sum([prices[item] * quantity for (item, quantity) in purchase])
```

Output:

```
220
```

Neat isn't it?

Chapter 29. Comments

You might be smart enough to write a piece of code today, but you may not be smart enough 6 months from now. The very code you created will look foreign and non-understandable. So it better for one to comment or add descriptions to ones code. All programming languages provide a way to write comments in them. Comments are there for programmers reference, when the program runs, the comments are ignored as the computer does not need it to run the code.

Take a look at the program below:

```
# Temperature in Celsius
C = 30

#=
The following code below calculates the Fahrenheit
from Celsius
=#
F = ((9 / 5) * C) + 32

println(30, " C = ", F, " F")
```

It's a simple program to convert Celsius to Fahrenheit. But notice here:

```
# Temperature in Celsius
C = 30
```

Here the things followed by the hash `, that is  Temperature in Celsius` is a comment. In the above code I have given `C = 30`, in reality it would have been far better if I had given `celsius = 30` as in today's way of coding we expect a code to be as good to read as a comment. Any way I hope you now get an idea what an comment is.

The example shown above is a single line comment. I could have given it like this too:

```
C = 30 # Temperature in Celsius
```

Personally I prefer comments being in a separate line.

Now if you want to write long paragraphs, there is a thing called multiline comments. It starts with a `= and ends with a =` as shown below:

```
#=
The following code below calculates the Fahrenheit
from Celsius
=#
F = ((9 / 5) * C) + 32
```

You can find the source file for the program here https://gitlab.com/datascience-book/code/-/blob/master/comments.jl.

Chapter 30. Functions

Let's say you go to a hotel, you order the waiter a dish and you get it. Just saying "I want Briyani" gets you something. No need to tell the waiter how to purchase mutton, rice cook them and all those stuff, just a few words and you are able to accomplish this complex task. Just imagine what you have achieved! There is no need for you to tell about how to water the fields and dry the paddy, raise goats and all those stuff, just a few words and you are done. All is taken care at the background.

Programming is very similar to real world. You can wrap lot of things into a code block, give it a name and call it when needed. Those things are called functions. So lets see a piece of code that makes us understand it a bit. Type the code below and execute it.

```
function printline()
    println('*' ^ 50)
end

printline()
println("Hello World!")
printline()
```

Output:

```
**************************************************
Hello World!
**************************************************
```

So notice that how a simple and readable is this function call `printline()`, and it prints us a set of stars for us. You can remember it, call it easily and its intuitive. Those are the power of functions.

When designing a function have these in mind:

- It's name should be intuitive and readable.
- A very good code needs almost no documentation.
- If the code is not understandable, then make sure you [comment]({% post_url 2020-11-26-comments %}) them well so that they will be understandable by the reader.
- If for some reason reading the function name is not obvious, say you think `printline()` may confuse some one who's native is not English, try to break it to `print_line()`. That is use underscore to break complex words.
- The code inside functions should be readable as well (we will talk about this further in future).

Okay now we know function is called by its name followed by round braces like `printline()` as shown above. Now let's see how its defined:

```
function printline()
    println('*' ^ 50)
end
```

You see above the keyword `function`, that says to Julia that we are defining a function. This is followed by the function name `printline` like this:

```
function printline
```

This is immediately followed by braces:

```
function printline()
```

So these define the start of the function, now let's say where it ends here using the `end` keyword:

```
function printline()
end
```

Now between the `function printline()` and `end`, we can fill it with code the function should execute as shown:

```
function printline()
    println('*' ^ 50)
end
```

In the above case we just tell it to print a line made up of 50 stars.

You can get the Jupyter notebook for this blog here https://gitlab.com/datascience-book/code/-/blob/master/functions.ipynb.

30.1. Passing Arguments

Let's come back to the hotel example. While ordering Briyani, you can pass some options to the waiter. Say you say that you do not need onions as side dish but want more Brinjal, then depending on the options your side dish will be tailored. These passing values to a function is technically in programming is called arguments.

Type the program below and execute it:

```
function printline1(length)
    println('*' ^ length)
end

i = 1

while i <= 10
    printline1(i)
    i += 1
end
```

Output:

```
*
**
***
****
*****
******
*******
********
*********
**********
```

So in the above program in these lines:

```
while i <= 10
    printline1(i)
    i += 1
end
```

We call `printline1` like this `printline1(i)` with an argument. This `i` is passed to variable called `length` and is available inside the body of the `printline1` function. With the `length` we vary the number of stars in the line as shown in below code:

```
function printline1(length)
    println('*' ^ length)
end
```

The Jupyter notebook for this blog is available here https://gitlab.com/datascience-book/code/-/blob/master/functions.ipynb.

30.2. Default Argument

Okay in `printline1()` function you saw that you can change the length of the line with an argument,

but you were left with calling the function with the argument always been set to vary the length of the line. Let's say what if you want to have a function where you can pass an argument, else the function assumes something by default. Welcome to default argument. Look at the program below:

```
function printline2(length = 50)
    println('*' ^ length)
end

printline2(10)
printline2()
```

Output:

```
**********
**************************************************
```

Here in the above example, function named `printline2()` is coded as follows: `function printline2(length = 50)`, note the `length = 50`. So you can call `printline2` like this `printline2(10)`, where it will print a line of 10 characters long, or you can call it like `printline2()` where it will print a line of 50 characters long which is the default provided.

The Jupyter notebook for this blog is available here https://gitlab.com/datascience-book/code/-/blob/master/functions.ipynb.

30.3. Default Argument

Okay in `printline1()` function you saw that you can change the length of the line with an argument, but you were left with calling the function with the argument always been set to vary the length of the line. Let's say what if you want to have a function where you can pass an argument, else the function assumes something by default. Welcome to default argument. Look at the program below:

```
function printline2(length = 50)
    println('*' ^ length)
end

printline2(10)
printline2()
```

Output:

```
**********
**************************************************
```

Here in the above example, function named `printline2()` is coded as follows: `function printline2(length = 50)`, note the `length = 50`. So you can call `printline2` like this `printline2(10)`,

where it will print a line of 10 characters long, or you can call it like `printline2()` where it will print a line of 50 characters long which is the default provided.

The Jupyter notebook for this blog is available here https://gitlab.com/datascience-book/code/-/blob/master/functions.ipynb.

30.4. More default arguments

In the previous blog we have seen functions with default argument, now let's see more of it. See the below program, type it and execute it:

```
function printline3(length = 50, character = '*')
    println(character ^ length)
end

printline3()
printline3(10)
printline3(20, '#')
```

Output:

```
**************************************************
**********
####################
```

You see a function that's starts with a definition like this `function printline3(length = 50, character = '*')`, and you see you can call it in three ways like without any argument like this:

```
printline3()
```

where the `length` defaults to `50` and `character` defaults to `'*'`. You can call it with a single argument like this:

```
printline3(10)
```

where the `length` becomes `10` but the character defaults to `'*'`. You can call it like this:

```
printline3(20, '#')
```

Where we vary both `length` and `character` away from the default values. But what if you want a function where you just vary the `character` alone so that it can be called like `printline3('>')`, currently it cannot be done. So look at the new function definition of `printline4` below, type it and execute it.

```
function printline4(length = 50, character = '*')
    println(character ^ length)
end

function printline4(char::Char)
    printline4(50, char)
end


printline4()
printline4(30)
printline4('#')
printline4(25, '>')
```

Output:

```
**************************************************
******************************
##################################################
>>>>>>>>>>>>>>>>>>>>>>>>>
```

Note the calls `printline4()`, `printline4(30)` and `printline4(25, '>')` are taken care by this definitions:

```
function printline4(length = 50, character = '*')
    println(character ^ length)
end
```

Now for this `printline4('#')`, we define a another `printline4()` function like this:

```
function printline4(char::Char)
    printline4(50, char)
end
```

This function takes in a character as argument, and it intelligently passes it to `printline4(50, char)`. So you can learn few things from these, that is:

- A function can call another function. `printline4(char::Char)` calls `printline4(length = 50, character = '*')` in the above case.
- A function with same name can have multiple definitions.
- Julia calls the reads an executes the appropriate definition depending on the type and number of arguments passed.

The Jupyter notebook for this blog is available here https://gitlab.com/datascience-book/code/-/blob/master/functions.ipynb.

30.5. Returning Values

Lets say you have a mathematical function $f(x) = 2x + 3$, you expect $f(5)$ to return 13. In fact functions in programming like in mathematics are designed to return something. Julia functions returns the output of the last statement by default. Let's see an example, type the example below and execute it:

```
function add(a, b)
    a + b
end


sum = add(5, 3)
println(sum)
```

Output:

```
8
```

In the above example, `println(sum)` prints 8, but how? Because `sum` is assigned to this `add(5, 3)`, that means `add(5, 3)` seems to have done something and returned it out. Now let's look at the definition of `add()` function:

```
function add(a, b)
    a + b
end
```

So it just consists of one statement `a + b` and that's the last statement in the function, so the computed value of `a + b` must have been returned out which would have been stored in `sum` and that's what gets printed.

This is in fact amazing thing. As we have seen few blogs before, going to a hotel and ordering a dish abstracts many things. You are served a dish, but behind it a cook works on it, before that a farmer would have produced something, before that a factory would have produced seeds fertilizers, animal feed, machinery etc, all of these are been abstracted away by a simple order where you say I want such and such a dish. Functions gives you that mighty power.

You could also specify the keyword `return` in a function to return any thing. In the example below:

```
function add_with_return(a, b)
    return a + b
end


sum = add_with_return(5, 3)
println(sum)
```

Output:

```
8
```

we explicitly specify `return a + b` so that the `a + b` gets returned. Now it does not mean its only at the last statement you must return something. Look at the code below:

```
function add_with_wrong_return(a, b)
    return 0
    return a + b
end


sum = add_with_wrong_return(5, 3)
println(sum)
```

Output:

```
0
```

in it we have returned 0 using `return 0` before `return a + b`, so no matter what ever you do, say `add(1, 2)` or what ever, it will return `0`. Once a function has returned something the execution of the function will stop, the code after return wont be executed, so in the above case `return a + b` is a unreachable code. In some IDE's and tooling environments,it would warn of possible unreachable code thus making you to code better.

The Jupyter notebook for this blog is available here https://gitlab.com/datascience-book/code/-/blob/master/functions.ipynb.

30.6. Named Arguments

Till now in functions you have seen functions where arguments are positional, say you define a function like this:

```
function sub(a, b)
  return a - b
end
```

here it's always `b`, the second argument that will get subtracted from `a`, the first argument. Like when you call `sub(5, 3)` that you will get 2 as output, but what if I want to do something like this `sub(b = 20, a = 70)`, where I expect output to be 70 minus 20 that is 50, no that isn't possible till now. Now let's see how to do these kind of stuff.

Code the function below in your Jupyter lab and let's see how it works.

```
function increment(number = 0; inc = 0, dec = 0)
    number + inc - dec
end
```

Output:

```
increment (generic function with 4 methods)
```

First, we call just `increment()` as shown below:

```
increment()
```

Output:

```
0
```

in this case the `number`, `inc` and `dec` defaults to `0`. And hence we get the output as `0`. Now let's call it with just one argument as shown:

```
increment(1)
```

Output:

```
1
```

here `increment(1)`, the `1` is passed as positional argument and hence `number` in `function increment(number = 0; inc = 0, dec = 0)` becomes `1` and hence `1` is returned.

Now let's see the code below:

```
increment(2, inc = 5)
```

Output:

```
7
```

and this code:

```
increment(12, dec = 8)
```

Output:

```
4
```

Looks like, `inc` and `dec` are not passed as positional argument, but they were named as `inc` in `increment(2, inc = 5)` and `dec` in `increment(12, dec = 8)`, yet Julia seems to have compute the results correctly. If these were positional then `8` should have been pass to `inc` within the function when we call `increment(12, dec = 8)`.

In the example below we give both `inc` and `dec` as named arguments to the function:

```
increment(10, inc = 5, dec = 7)
```

Output:

```
8
```

So how Julia knows `inc` and `dec` are named arguments but not positional. If you see the function definition `function increment(number = 0; inc = 0, dec = 0)`, you can see we have place `inc = 0` and `dec = 0` after a semicolon `;`, that's a hint to Julia that these arguments could be named arguments.

Just because we have said that they are named arguments, it does not mean that they have lost their positional status. We can very well call the `increment()` function as shown below, where every argument preserves its positional properties.

```
increment(10, 5, 7)
```

Output:

```
8
```

Now see how I have coded the function `printline5()` below, where all arguments are placed after the semicolon `;` and are hence named arguments, and they have default values too:

```
function printline5(;length = 50, character = '*')
    println(character ^ length)
end
```

Output:

```
printline5 (generic function with 1 method)
```

So if I want a line of variable length I can call as shown:

```
println5(length = 7)
```

```
*******
```

If I want a line of different character, I can call like this:

```
println5(character = '@')
```

Output:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
```

Let's say I want a line of my preferred length and the character of my choice, I can call like this:

```
println5(length = 7, character = '!')
```

Output:

```
!!!!!!!
```

And I can call it with out any argument at all, in the case below it takes default `length` and `character`:

```
println5()
```

Output:

```
**************************************************
```

But since all are placed behind a semicolon, I notice that they have lost their positional status in this case:

```
println5(7, '!')
```

Output:

```
MethodError: no method matching printline5(::Int64, ::Char)



Stacktrace:

 [1] top-level scope at In[53]:1

 [2] include_string(::Function, ::Module, ::String, ::String) at ./loading.jl:1091

 [3] execute_code(::String, ::String) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:27

 [4] execute_request(::ZMQ.Socket, ::IJulia.Msg) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/execute_request.jl:86

 [5] #invokelatest#1 at ./essentials.jl:710 [inlined]

 [6] invokelatest at ./essentials.jl:709 [inlined]

 [7] eventloop(::ZMQ.Socket) at
/home/karthikeyan/.julia/packages/IJulia/a1SNk/src/eventloop.jl:8

 [8] (::IJulia.var"#15#18")() at ./task.jl:356
```

Looks like you need one positional argument before the semicolon `;` for named argument to preserve their positional properties.

You can get the Jupyter notebook file for this blog here https://gitlab.com/data-science-with-julia/code/-/blob/master/functions.ipynb.

30.7. Single line functions

There are more tricks you can do with functions in Julia. If the function is short, you can simply define it as shown:

```
y(x) = 2x + 3
```

Output:

```
y (generic function with 1 method)
```

Noe we call `y(7)` as shown below:

```
y(7)
```

Output:

```
17
```

Doesn't it look very mathematical? That's why many math people use Julia and I feel its better for A.I, as A.I is nothing but math.

30.8. Functions acting on a vector

Do you know that a function can operate on a vector? We use the same function `y()` defined above on a vector, see the code below:

```
y.([1, 2, 3, 4])
```

Output:

```
4-element Array{Int64,1}:
  5
  7
  9
 11
```

See how `y()` just takes single argument `x` and returns a value, but `y.()` that is with a dot can take a vector, operate on it element wise one by one, pack it into a vector and return it out. It's not that functions in a single line can do it, any function can do it as shown below:

```
function y1(x)
    2x + 3
end
```

Output:

```
y1 (generic function with 1 method)
```

```
y1.([1, 2, 3, 4])
```

Output:

```
4-element Array{Int64,1}:
  5
  7
  9
 11
```

And this function with a `.` can operate on ranges too:

```
y.(1:10)
```

Output:

```
10-element Array{Int64,1}:
  5
  7
  9
 11
 13
 15
 17
 19
 21
 23
```

30.9. Using functions with map

There is a function called `map()` which takes function as argument and iterateable thing as second argument, say `Array`, `Tuple` or `Range`, then it takes each element in the iterator, apply the function on it, takes the result, packs it into an array and returns it to us.

Take a look at the code below, see how we map `y` on to a `Array`

```
map(y, [1, 2, 3, 4])
```

Output:

```
4-element Array{Int64,1}:
  5
  7
  9
 11
```

30.10. Anonymous function

It is not that a function should always have a name, it could be anonymous too, take a look at the code below, type it and execute it:

```
map(x -> 2x + 3, [1, 2, 3, 4])
```

Output:

```
4-element Array{Int64,1}:
  5
  7
  9
 11
```

We have `map()`, rather than giving a name of a function as the first argument, we give `x → 2x + 3`, that is the function has no name, but an argument `x` followed by arrow `→` followed by the return value `2x + 3`, and it works like charm.

30.11. Variable Arguments

Its not that there must be fixed set of arguments that are passed to a function, you can pass variable number of arguments too as shown, type the function below in a Jupter lab cell:

```
function vararg_sum(numbers...)
    total = 0
    for number in numbers
        total += number
    end
    total
end
```

Output:

```
vararg_sum (generic function with 3 methods)
```

So a function was created, now let's use it with a single argument:

```
vararg_sum(8)
```

Output:

```
8
```

So it works. Now let's try many arguments:

```
vararg_sum(8, 2, 3, 4, 5)
```

Output:

```
22
```

It works too! We will soon see how it works.

A variable that gets variable argument is at the end of the function. It has a name `numbers` in out case, and is followed by a triple dot `⋯` as in `function vararg_sum(numbers⋯)`. Inside the function the variable named `numbers` and not `numbers⋯` stores all the argument. We will see how it stores soon. Julia gets a hint hat a variable need to store varible arguments if its followed by triple dot `⋯`.

A variable argument variable should always be at the end of the function if we are using other variables as named and positional arguments. The below function find total of `numbers` and adds it with a base `base`. As you can see the variable argument is at the end of the function.

```
function vararg_sum_with_base(base = 0, numbers...)
    total = base
    for number in numbers
        total += number
    end
    total
end
```

Output:

```
vararg_sum_with_base (generic function with 2 methods)
```

and once again it works as shown below:

```
vararg_sum_with_base(5, 1, 2, 3, 4)
```

Output:

```
15
```

In the below example, all we are interested is to see what `numbers` hold. So let's print it out:

```
function vararg(base = 0, numbers...)
    total = base
    println(numbers)
    println(typeof(numbers))
end
```

Output:

```
vararg (generic function with 2 methods)
```

We call the function

```
vararg(1, 2, 3, 4, 5)
```

```
(2, 3, 4, 5)
NTuple{4,Int64}
```

As you see `number` inside the function is nothing but a `Tuple` packed with all the values we pass as argument.

30.12. Piping / Chaining functions

Julia has this nice idea of chaining functions, the below code is equivalent to `sum(1:10)`:

```
1:10 |> sum
```

Output:

```
55
```

So in the above code we see that `1:10` is piped to `sum()`. The output of this can be piped to `sqrt()` as shown below:

```
1:10 |> sum |> sqrt
```

Output:

```
7.416198487095663
```

So the above code is equivalent to `sqrt(sum(1:10))`, but it looks more elegant. The same operation is

done by code below using the circle operator, you can type it by typing `\circ` and pressing `Tab` in your notebook:

```
(sqrt ∘ sum)(1:10)
```

Output:

```
7.416198487095663
```

Notice how in piping things pass from left to right here `1:10 |> sum |> sqrt` and right to left here `(sqrt ∘ sum)(1:10)`.

So these are the useful things about functions I feel we as data scientist must know.

You can get the Jupyter notebook file for this blog here https://gitlab.com/datascience-book/code/-/blob/master/functions.ipynb.

30.13. Passing function as argument

Video lecture for this section could be found here https://youtu.be/fNz2IlpG41A

It is possible in Julia to pass an function as argument, below we define a function called `starline()` that prints a line of 50 stars:

```
starline() = println('*' ^ 50)
```

Output:

```
starline (generic function with 1 method)
```

Let's test it:

```
starline()
```

Output:

```
**************************************************
```

Now let's define a function called `execute_function()` as follows:

```
function execute_function(a_function)
    a_function()
end
```

Output:

```
execute_function (generic function with 1 method)
```

So how does this `execute_function()` works. First we take in an argument as `a_function` and in this statement `a_function()`, we are treating it as a function and calling it. Now let's test it:

```
execute_function(starline)
```

Output:

```
**************************************************
```

So it works. You can find notebook here https://gitlab.com/datascience-book/code/-/blob/master/passing_function_as_arguments.ipynb.

30.14. Multiple Dispatch

Video lecture for this section could be found here https://youtu.be/fNz2IlpG41A

Take a look at the code below, type it and execute it in your Jupyter notebook or in a text file. Let's see how it works. If you see, we have defined a functions named `multi_disp` many times:

```
function multi_disp()
    println("In multi_disp()")
end

function multi_disp(some_argument)
    println("In multi_disp(some_argument)")
end

function multi_disp(number::Int)
    println("In multi_disp(number::Int)")
end

function multi_disp(arg1, arg2)
    println("In multi_disp(arg1, arg2)")
end


multi_disp()
multi_disp("abc")
multi_disp(70)
multi_disp(1, 2)
```

Output

```
In multi_disp()
In multi_disp(some_argument)
In multi_disp(number::Int)
In multi_disp(arg1, arg2)
```

So if you see this piece of code

```
multi_disp()
multi_disp("abc")
multi_disp(70)
multi_disp(1, 2)
```

one can see `multi_disp()` calling this function definition:

```
function multi_disp()
    println("In multi_disp()")
end
```

Notice that the function definition has no arguments, and `multi_disp()` has no arguments too!

Now take for instance we are calling `multi_disp("abc")`, some how this executes this function:

```
function multi_disp(some_argument)
    println("In multi_disp(some_argument)")
end
```

and not any other function link this one:

```
function multi_disp(number::Int)
    println("In multi_disp(number::Int)")
end
```

Julia knows that `function multi_disp(number::Int)` takes an integer for argument and hence should be avoided when we call `multi_disp("abc")` where `"abc"` is a string. But when `multi_disp(70)` is called, Julia rightly executes `multi_disp(number::Int)` because we are passing in number. This is called multiple dispatch, and it is one of the most powerful feature that Julia offers to programmers. We will see it when we create complex programs where a same function needs to do multiple things depending on number of arguments and types of arguments varies.

Now for `multi_disp(1, 2)`, Julia rightly calls:

```
function multi_disp(arg1, arg2)
    println("In multi_disp(arg1, arg2)")
end
```

Because the above function has two arguments. So Julia takes into account the number of arguments and the arguments types when trying to decide which function to call when function names are same.

Get the notebook for this blog here https://gitlab.com/datascience-book/code/-/blob/master/multiple_dispatch.ipynb.

Chapter 31. Regular Expressions (regexp)

Video lecture for this section could be found here https://youtu.be/AkYyjhEizFA

Get the Jupyter notebook here https://gitlab.com/datascience-book/code/-/blob/master/regexp.ipynb

Regular expressions are used to check if a pattern exists in a string. Without much blah blah let's see what it is.

31.1. A taste of Regexp

Type the code below and execute

```
regexp = r"abc"
m = match(regexp, "english letters start with abc")
```

Output:

```
RegexMatch("abc")
```

Take this statement:

```
regexp = r"abc"
```

Here you have a variable named `regexp` which is been assigned to `r"abc"`, which is not a string. Even thought the `abc` are surrounded by double quotes, the `r` before the quotes makes it an expression.

Now in this line:

```
m = match(regexp, "english letters start with abc")
```

Julia see's if the expression (not string) `abc` is present in the string `"english letters start with abc"`, and it is present it returns a match, and hence we get the output `RegexMatch("abc")`.

In the example below, you see that circular brackets surround `abc` and make it `(abc)`. This means we want to capture that expression, let's executeand see what happens:

```
regexp = r"(abc)"
m = match(regexp, "english letters start with abc")
```

Output:

```
RegexMatch("abc", 1="abc")
```

So we get the output that it matches `abc`, which is denoted by `RegexMatch("abc"`, and this is followed by capture which is denoted by `1="abc")`.

Now you can get this match is captured in variable `m` and you can get it as shown below:

```
m[1]
```

Output:

```
"abc"
```

These things are called captures.

`\d` is used to match a digit. In the example below string `"Five means 5 in English."` contains digit `5` and so `match` returns a positive match:

```
regexp = r"\d"
m = match(regexp, "Five means 5 in English.")
```

Output:

```
RegexMatch("5")
```

Now let's say we wants to match exactly 6 digits. We know `\d` means match a digit, and we append `{6}` to it to match exactly 6 times as shown below:

```
regexp = r"\d{6}"
m = match(regexp, "I live in 36/1, my postal code is 600131.")
```

Output:

```
RegexMatch("600131")
```

In the above example `36` consists of digits and so does `1` in `36/1`, they don't match `\d{6}` because they are two and one digit each, the one that matches is `600131`.

The plus sign denotes one or more so `\d+` means we need to match one or more digit. So the below example matches `36` in `"I live in 36/1, my postal code is 600131."`:

```
regexp = r"(\d+)"
m = match(regexp, "I live in 36/1, my postal code is 600131.")
```

Output:

```
RegexMatch("36", 1="36")
```

since we have given it like `(\d+)` which means capture, we get the output as `RegexMatch("36", 1="36")`.

The example below shows how to check for a match:

```
regexp = r"(\d{6})"
m = match(regexp, "I live in 36/1, my postal code is 600131.")

if m != nothing
    println("There is a match")
end
```

Output:

```
There is a match
```

`600131` is a match for `(\d{6})`, so there is a match. You can check if there is a match by capturing it in a variable, in this case its `m`, and if there is a match it should not be `nothing`. So any thing inside this `if` block:and

```
if m != nothing
    println("There is a match")
end
```

Get's executed if there is a match, so the code prints `There is a match`.

In the code below we check for a 6 digit string which is not present in `"I live in 36/1, my postal code is 600."`, and hence the condition `m != nothing` becomes `false` and so nothing gets printed.

```
regexp = r"(\d{6})"
m = match(regexp, "I live in 36/1, my postal code is 600.")

if m != nothing
    println("There is a match")
end
```

The code below is same as the example above, but here we have got the `else` part, since `m !=`

`nothing` becomes `false`, the else part gets executed and `Match not found` gets printed.

```
regexp = r"(\d{6})"
m = match(regexp, "I live in 36/1, my postal code is 600.")

if m != nothing
    println("There is a match")

    else

    println("Match not found")
end
```

Output:

```
Match not found
```

Hope you have got some vague idea about regexp, we will see it in bit more detail in the coming sections.

31.2. Things to remember

There are some things you need to remember, or at least refer from time to time. Those are mentioned in table below[10].

If you do not understand now, don't worry, you will be able to pick it up.

Thing	What it means
.	Any single character
\w	Any word character (letter, number, underscore)
\W	Any non-word character
\d	Any digit
\D	Any non-digit
\s	Any whitespace character
\S	Any non-whitespace character
\b	Any word boundary character
^	Start of line
$	End of line
\A	Start of string
\z	End of string
[abc]	A single character of: a, b or c

Thing	What it means
[^abc]	Any single character except: a, b, or c
[a-z]	Any single character in the range a-z
[a-zA-Z]	Any single character in the range a-z or A-Z
(…)	Capture everything enclosed
(a	b)
a or b	a?
Zero or one of a	a*
Zero or more of a	a+
One or more of a	a{3}
Exactly 3 of a	a{3,}
3 or more of a	a{3,6}
Between 3 and 6 of a	i
case insensitive	m
make dot match newlines	x
ignore whitespace in regex	o

31.3. The dot

If you refer the table, the dot `.` in regexp is used to match anything. Take a regexp `r".at"`, the `.at` means that it will match anything that's before `at`, so it matches `rat` in `There is rat in my house"`:

```
match(r".at", "There is rat in my house")
```

Output:

```
RegexMatch("rat")
```

It matches `cat` in `"There is cat in my house"`:

```
match(r".at", "There is cat in my house")
```

Output:

```
RegexMatch("cat")
```

It matches `bat` in `"There is bat in my house"`

```
match(r".at", "There is bat in my house")
```

Output:

```
RegexMatch("bat")
```

31.4. Character classes

The dot before at `.at`, matches anything before `.at`, but let's say we want to match only, `b`, `c` and `r` before `at`, we can group them as character class like this `[bcr]`, where we put them inside square braces, now we follow it with at like this `[bcr]at`, this matches `bat`, `cat`, `rat`, but notthing else.

Take a look at the examples below where this regular expression `r"[bcr]at"` is put into action and matches stuff:

```
match(r"[bcr]at", "There is rat in my house")
```

Output:

```
RegexMatch("rat")
```

```
match(r"[bcr]at", "There is cat in my house")
```

Output:

```
RegexMatch("cat")
```

```
match(r"[bcr]at", "There is bat in my house")
```

Output:

```
RegexMatch("bat")
```

But in the example below, we have a string that has no bat, cat or rat in it, so it returns nothing:

```
match(r"[bcr]at", "There is mat in my house")
```

Below are simple programs where regexp `r"[bcr]at"` is used to detect presence of the animal names bat, cat or rat, if yes it prints `here is an animal in the house.` else it prints `I don't detect an`

`animal.`.

```
string = "There is rat in my house"

match_data = match(r"[bcr]at", string)

if match_data != nothing
    println("There is an animal in the house.")
else
    println("I don't detect an animal.")
end
```

Output:

```
There is an animal in the house.
```

```
string = "There is mat in my house"

match_data = match(r"[bcr]at", string)

if match_data != nothing
    println("There is an animal in the house.")
else
    println("I don't detect an animal.")
end
```

Output:

```
I don't detect an animal.
```

We can provide something like ranges in character classes, say `[A-Z]` means match anything from capital `A` to capital `Z` (both inclusive), so in the example below `U` is the first match, so it get's detected:

```
match(r"[A-Z]", "this string contains UPPERCASE letter")
```

Output:

```
RegexMatch("U")
```

There is no capital letter in `"this string does not contains uppercase letter"`, so the below example returns nothing as a match:

```
match(r"[A-Z]", "this string does not contains uppercase letter")
```

In computers `A` to `Z` are ordered before `a` to `z`, so if we give a regular expression `r"[A-z]"` as show below:

```
match(r"[A-z]", "There is a letter")
```

Output:

```
RegexMatch("T")
```

It detects any letter, so the first letter `T` gets matched and we get `RegexMatch("T")` as output. How ever this same regexp does not match numbers, so we get nothing as out put in the program below:

```
match(r"[A-z]", "12345")
```

If we want to match numbers, we can use `\d` or in case of character classes we can use `[0-9]` as shown below to return a positive match:

```
match(r"[0-9]", "12345")
```

Output:

```
RegexMatch("1")
```

In character classes `^` (carrot) signifies not, so when we say `[^0-9]` it means detect any thing that not 0, 9 and and any thing in between, so the string in example below `"12345"` is just numbers, so nothing gets detected.

```
match(r"[^0-9]", "12345")
```

How ever the same regep detects `a` in the string `"12345 abc"` below because its not a number:

```
match(r"[^0-9]", "12345 abc")
```

Output:

```
RegexMatch(" ")
```

31.5. Anchors

Let's say you want to match something at the start and end of a string, you can use a thing called anchors, once again refer to things to remember section for this. Anchor's `\A` and `^` are used to denote start of a string, hence `r"\AHello"` and `r"^Hello"` checks if Hello exists at the start of the string, if yes it matches as shown in the below examples:

```
match(r"\AHello", "Hello world!")
```

Output:

```
RegexMatch("Hello")
```

```
match(r"^Hello", "Hello world!")
```

Output:

```
RegexMatch("Hello")
```

In the below example we don't have a match because `Hello` is not at the start of `"Say Hello world!"`:

```
match(r"\AHello", "Say Hello world!")
```

Similarly anchors `\Z` and `$` are used to match something that should be present at the end of a string, so look at the exples below and explain it to yourself:

```
match(r"world!\Z", "Hello world!")
```

Output:

```
RegexMatch("world!")
```

```
match(r"world!$", "Hello world!")
```

Output:

```
RegexMatch("world!")
```

In the example below, `world!` is not present at the end of `"Hello world!, said the computer"` , so

there is no match and it returns `nothing`.

```
match(r"world!\Z", "Hello world!, said the computer")
```

31.6. Captures

Regexp is used to detect a pattern in a string, but what if we want to see what string it had matched? Welcome to captures. Captures are created by wrapping regexp in round braces as shown:

```
match(r"(abc)", "This string contains abc in it")
```

Output:

```
RegexMatch("abc", 1="abc")
```

In the above example we have wrapped `abc` in regexp like this: r"(abc)", so apart from returning a positive match `RegexMatch("abc")`, it will also return a capture.

Look at the captures below, first we capture `abc` like this: `(abc)`, inside the capture we once again capture `bc` ike this `(a(bc))`, execute the code:

```
match(r"(a(bc))", "This string contains abc in it")
```

Output:

```
RegexMatch("abc", 1="abc", 2="bc")
```

Regexp start from the outside, so first it captures `abc` in `"This string contains abc in it"` and then it captures `bc` inside the first capture. So you end up with two captures like this: `RegexMatch("abc", 1="abc", 2="bc")`.

Look at the example below, execute it:

```
match(r"(a(b(c)))", "This string contains abc in it")
```

Output:

```
RegexMatch("abc", 1="abc", 2="bc", 3="c")
```

Let's see how it works. So we have a capture as shown: `(a(b(c)))`. Forget the inner braces, so the outer most braces looks like `(abc)`, so `abc` is captured and we get `RegexMatch("abc", 1="abc")`. Next

come to the inner braces that captures `bc (a(bc))`, so that become the second capture and we get `RegexMatch("abc", 1="abc", 2="bc")`. Now the most inner most capture is just `c` as in `(a(b(c)))`, so just `c` is in third capture and we get `RegexMatch("abc", 1="abc", 2="bc", 3="c")`.

31.6.1. Capturing phone number

Let's look at something that might be practical. Let's say that one would write his or her phone number in this format: +91 8428050777, where +91 is the country code and 8428050777 is the phone number. Now we need to write a regexp that would capture it.

Look at the piece of code below and let's execute it:

```
match(r"((\d{2}) (\d{10}))", "+91 8428050777")
```

Output:

```
RegexMatch("91 8428050777", 1="91 8428050777", 2="91", 3="8428050777")
```

You need to refer Things to remember section in Regexp

So we have got first capture as the entire phone number. This is accomplished by the regular expression `r"(\d{2} \d{10})"`, where `r"(\d{2})"` matches the `91` in `"+91 8428050777"`, next there is a space, s the regexp is now `r"(\d{2}) "` and then there is a ten digit number , so its now `r"(\d{2} \d{10})"`. this would give a match and capture like this: `RegexMatch("91 8428050777", 1="91 8428050777")`

Next we need to capture the country code which is nothing but the first two digits in the regexp, so the regexp now becomes like this: `r"((\d{2}) \d{10})"`, notice the braces around `\d{2}`. So this would produce a match and capture like this `RegexMatch("91 8428050777", 1="91 8428050777", 2="91")`.

Similarly we apply a capture around `\d{10}` for local area number so we get a regexp as shown: `r"\d{2}) (\d{10}"`, this gives us our final match and capture as like this: `RegexMatch("91 8428050777", 1="91 8428050777", 2="91", 3="8428050777")`.

in the program below, we see how to use captures, execute it and we will see baout it.

```
match_data = match(r"((\d{2}) (\d{10}))", "+91 8428050777")

println("Phone Number: ", match_data[1])
println("Country code: ", match_data[2])
println("Local number: ", match_data[3])
```

Output:

```
Phone Number: 91 8428050777
Country code: 91
Local number: 8428050777
```

So the above code we assign `match(r"\d{2}) (\d{10}", "+91 8428050777")` to a variable named `match_data` in this line:

```
match_data = match(r"((\d{2}) (\d{10}))", "+91 8428050777")
```

So `match_data` contains `RegexMatch("91 8428050777", 1="91 8428050777", 2="91", 3="8428050777")`. Which means `match_data[1]` will have `"91 8428050777"`, `match_data[2]` will have `91` and `match_data[3]` will have the capture `"8428050777"`. We print those in the following lines of code:

```
println("Phone Number: ", match_data[1])
println("Country code: ", match_data[2])
println("Local number: ", match_data[3])
```

31.7. Counts

Refer Things to remember section to understand the code.

Looks at the piece of code below, the regex `r"\d"` matches exactly one digit that is `8` in `"8420050777"`.

```
match(r"\d", "8420050777") # match just one digit
```

Output:

```
RegexMatch("8")
```

Now say we want to match more than one digit, we could use the `+` sign after `\d` to get regular expression like `r"\d+"`. So this identifies a (sub)string having one or more digit as shown below:

```
match(r"\d+", "8420050777") # + means one or more
```

Output:

```
RegexMatch("8420050777")
```

So the whole number `8420050777` gets matched.

Now `*` in regex means zero or more and `\s` means space, so type the code below and execute it:

```
match(r"\w*\s*\d+", "Karthik 8420050777") # * zero or more
```

Output:

```
RegexMatch("Karthik 8420050777")
```

You must read it as zero or more printable characters `r"\w*"`, followed by zero or more spaces `r"\w*\s*"`, followed by one or more numbers `r"\w*\s*\d+"`. So the above example matches "Karthik 8420050777".

You can remove the name, increase or decrease spaces, and it would still work. I have removed the name in the example below and the above regex works as shown:

```
match(r"\w*\s*\d+", "8420050777") # * zero or more
```

Output:

```
RegexMatch("8420050777")
```

in the next example , we will try to form regular expression where it needs to match something like `"Karthik: 8420050777"`. The example below fails to produce a math because there is no colon in the string.

```
match(r"\w*: \d+", "Karthik 8420050777")
```

To fix it we use the zero or one thing, which is denoted by `?` as shown below:

```
match(r"\w*:? \d+", "Karthik 8420050777") # ? means zero o one
```

Output:

```
RegexMatch("Karthik 8420050777")
```

So the regex should be read as zero or more words/printable characters `r"\w*"`, followed by zero or one colon `r"\w*:"`, followed by exactly one space `r"\w*:? "`, followed by one or more numbers.

This would even match when there is a colon in between name and space as shown below:

```
match(r"\w*:? \d+", "Karthik: 8420050777")
```

Output:

```
RegexMatch("Karthik: 8420050777")
```

Now let's capture the name and phone number:

```
match(r"(\w*):? (\d+)", "Karthik: 8420050777")
```

Output:

```
RegexMatch("Karthik: 8420050777", 1="Karthik", 2="8420050777")
```

To make the regex more robust, let's introduce zero or more spaces between the `:` and did it as shown below:

```
match(r"(\w*):?\s*(\d+)", "Karthik: 8420050777")
```

Output:

```
RegexMatch("Karthik: 8420050777", 1="Karthik", 2="8420050777")
```

Note in the above example we have added `\s*` between `:?` and `\d+`.

Now let's test our almost bullet proof regex with lot of spaces between name and number:

```
match(r"(\w*):?\s*(\d+)", "Karthik:     8420050777")
```

Output:

```
RegexMatch("Karthik:     8420050777", 1="Karthik", 2="8420050777")
```

Let's say t hat you want to match exact number of counts, say only 4 numbers, you could use something lke this:

```
match(r"\d{4}", "Karthik:     8420050777")
```

Output:

```
RegexMatch("8420")
```

Since we seen usage of `{<number>}` in other sections in Regex, I am not going to go more into this.

31.8. String to regexp

May be one day you need to write a piece of code where you need to construct a regular expression dynamically. For that you need to construct a string, and this needs to be used in a regexp.

So look at the code below and execute it:

```
string = "\\d{4}"
regex = Regex(string)
```

Output:

```
r"\d{4}"
```

When wen print the above string using `println(string)` it prints `\d{4}` [11], we pass `string` to a function called `Regex` like this: `Regex(string)` and it gets converted to a regular expression. We do capture it in a variable named `regex`.

Now let's use the regexp to amtch some numbers:

```
match(regex, "43248")
```

Output:

```
RegexMatch("4324")
```

It works!

Below is a code where I have use `repr` function to convert regex into string, it was done for curiosity. May be it could serve you some purpose some day:

```
regexp = r"\d+"
println(repr(regexp))
```

Output:

```
r"\d+"
```

31.9. Case sensitive & insensitive match

We can tell a regexp to do case sensitive or insensitive match, for example take the example below:

```
match(r"(?i)uppercase", "This matches UPPERCASE")
```

Output:

```
RegexMatch("UPPERCASE")
```

`r"uppercase"` should not match `UPPERCASE` but it can be made to match it by adding `(?i)` to the prior of `uppercase`, so it becomes like this `r"(?i)uppercase`. This would match `UPPERCASE` in `"This matches UPPERCASE"`.

Since it's a case insensitive match, it also matches `uppercase` as shown below:

```
match(r"(?i)uppercase", "This matches uppercase")
```

Output:

```
RegexMatch("uppercase")
```

A very similar example is shown below, I think you should be able to explain it by now:

```
match(r"(?i)UPPERCASE", "This matches uppercase")
```

Output:

```
RegexMatch("uppercase")
```

`(?i)` switches on caseinsensivivity, wheras `(?-i)` switches it off. So look at the example below:

```
# Mixing case sensitivity
match(r"sensitive(?i)caseless(?-i)sensitive", "sensitiveCASELESSsensitive")
```

Output:

```
RegexMatch("sensitiveCASELESSsensitive")
```

`r"sensitive"` matches `sensitive`, then caseinsenivity gets switched on using `(?i)`, so `r"sensitive(?i)caseless"` matches `sensitiveCASELESS`, now we make it case sensitive using this switch `(?-i)`, and hence `r"sensitive(?i)caseless(?-i)sensitive"` matches the whole `sensitiveCASELESSsensitive`, and we get a match.

31.10. Scanning

You might be interested to scan a string for regexp and collect the words that match it, for that kind of operations I have written a function `scan_regexp` as shown below. Type the code and execute it, we will see how it works.

```
string = "I have bat, cat and rat in my house"

function scan_regexp(regexp, string)
    string = replace(string, r"\W" => " ")
    words = split(string)

    output = []

    for word in words
        if match(regexp, word) != nothing
            push!(output, word)
        end
    end

    output
end

scan_regexp(r".at", string)
```

Output:

```
3-element Vector{Any}:
 "bat"
 "cat"
 "rat"
```

First we start with an empty function like this:

```
function scan_regexp(regexp, string)
end
```

Let us receive the regular expression in variable `regexp` and the string that should be scanned in variable `string`.

Now if you refer Things to remember section, `\W` in regexp means non alphabetic character's, so we create a regular expression to scan all non alphabets and replace them with spaces in the following line `string = replace(string, r"\W" ⇒ " ")`.

```
function scan_regexp(regexp, string)
    string = replace(string, r"\W" => " ")
end
```

Next we split the string into words using the this line `words = split(string)`, so the function now becomes like this:

```
function scan_regexp(regexp, string)
    string = replace(string, r"\W" => " ")
    words = split(string)
end
```

Let's now have a array named `output` that will collect all `words` that match the `regexp`

```
function scan_regexp(regexp, string)
    string = replace(string, r"\W" => " ")
    words = split(string)

    output = []
end
```

Now for each word in word we compare if it matches the regular expression:

```
function scan_regexp(regexp, string)
    string = replace(string, r"\W" => " ")
    words = split(string)

    output = []

    for word in words
        if match(regexp, word) != nothing
        end
    end
end
```

If it matches we push that word into output using the following code: `push!(output, word)`, so the function now looks like this:

```
function scan_regexp(regexp, string)
    string = replace(string, r"\W" => " ")
    words = split(string)

    output = []

    for word in words
        if match(regexp, word) != nothing
            push!(output, word)
        end
    end
end
```

Finally we return the `output`:

```
function scan_regexp(regexp, string)
    string = replace(string, r"\W" => " ")
    words = split(string)

    output = []

    for word in words
        if match(regexp, word) != nothing
            push!(output, word)
        end
    end

    output
end
```

31.11. Learn more about regex

Regular expression is huge topic, we have just scratched the surface to give an introduction to you, but you need to be good with it if you are doing text mining and other operations. So I strongly encourage the reader to do more research, seek and learn about it.

You may find this gist to be useful https://gist.github.com/dataPulverizer/23c8d992d351d7faf0ed1c1966605b10.

You can also refer these books

1. Introducing regular expressions https://www.amazon.com/dp/B008K9OGDA
2. Mastering regular expressions https://www.amazon.com/dp/B007I8S1X0

[10] I got this list from http://rubular.com/

[11] One might need to learn about escape sequence in strings https://www.youtube.com/watch?v=dJ4PnBXkzXI

Chapter 32. Struct

Video lecture for this section could be found here https://youtu.be/Izsx3_IM81k

Get the Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/struct.ipynb

Let's say you want to find area of rectangle, you could write a Julia function named `rectangle_area(length, breadth)` that returns the rectangle area. But don't you think it will be intuitive if we can write something like this `area(rect::Rectangle)`, where the `rectangle` holds a special data type which packs `length` an `breadth` into one variable? Let's see hoe it can be done.

There is a thing called `struct` in Julia with which we can define complex data types. So in the code below we have created our own type called `Rectangle which can hold `length` and `breadth`. `

```
struct Rectangle
    length
    breadth
end
```

An instance of type `Rectangle` can be created as shown below:

```
rect = Rectangle(4, 5)
```

Output:

```
Rectangle(4, 5)
```

In the above piece of code `rect` variable holds an object of type `Rectangle`, this rect can be thought as a union of two variables namely `length` and `breadth`. Now let's write `area` function to find out the area of `Rectangle`.

The area function can be written as shown below.

```
function area(rect::Rectangle)
    rect.length * rect.breadth
end
```

If you see the function accepts one argument named `rect`, which must be of type `Rectangle`, else the function would throw an error. Inside the function we are returning the multiple of `rect's `length`, accessed by dot `.` operator like this: `rect.length` and it's `breadth`, which is accessed by dot operator like this `rect.breadth`.

Finally we can use our function arra, to calculate area of rectangle as shown below:

```
area(rect)
```

Output:

```
20
```

It works!

Now let's create a class called `Square`. A square can be defined just by its side length, so we just have one variable in it called side as shown below:

```
struct Square
    side
end
```

We can create an instance of `Square` as shown:

```
s = Square(5)
```

Output:

```
Square(5)
```

Now let's write area function for `Square` type. We write is as shown:

```
function area(square::Square)
    side = square.side
    area(Rectangle(side, side))
end
```

A square is nothing but rectangle having equal sides, so we can reuse `area(rect::Rectangle)`. So in the above function we get the side length of square using the following statement `side = square.side`, and we construct a rectangle from it using `Rectangle(side, side)`, and we call `Rectangle's area upon it like this: area(Rectangle(side, side)).

Now let's test it out:

```
area(s)
```

Output:

```
25
```

It works!

We can access attribute of a struct's instance using dot `.` operator as shown:

```
s.side
```

Output:

```
5
```

But what happens if we want to change the side of square:

```
s.side = 7
```

Output:

```
setfield! immutable struct of type Square cannot be changed
```

```
Stacktrace:
```

```
[1] setproperty!(x::Square, f::Symbol, v::Int64)
```

```
@ Base ./Base.jl:34
```

```
[2] top-level scope
```

```
@ In[11]:1
```

```
[3] eval
```

```
@ ./boot.jl:360 [inlined]
```

```
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
```

```
@ Base ./loading.jl:1094
```

It fails!

32.1. Mutable Struct

Le's say we want to define a data type called `Person` that stores ones name and age. Age changes, so it's better to have it as mutable struct. mutable struct is a struct who's instance variable's attributes can be reassigned to new values.

This is not a Julia performance book, so I do not want to go into pro's and cons of mutable and immutable things in computer programming. But if you really want this book to have such a content, do contact me, I might be tempted to write it.

So we create a mutable struct called `Person` as shown:

```
mutable struct Person
    name
    age
end
```

We create a person instance:

```
p = Person("Karthik", 38)
```

Output:

```
Person("Karthik", 38)
```

We can reassign the person age, or even the persons name, in the code below we are changing the person's age:

```
p.age = 39
```

Output:

```
39
```

Now let's print the person and see if the age has changed:

```
p
```

Output:

```
Person("Karthik", 39)
```

Now let's assign person's name to 7824 and age to Zigor:

```
p2 = Person(7824, "Zigor")
```

Ouput:

```
Person(7824, "Zigor")
```

It works! But it shouldn't. How can a person's age be Zigor. Let's fix such kind of things up.

32.2. Value Types

It turns out that in a struct we can tell what type the attribute should be. If you look at the code below, we create a struct named `Robot` and it has two attributes `name` and `model_number`. We enforce `name` to be of type string using the `name::String` statement, and we enforce `model_number` to be integer using `model_number::Int`.

```
struct Robot
    name::String
    model_number::Int
end
```

Now if try to create instance of robot using mismatched data types, we will get an error as shown:

```
r = Robot(7824, "Zigor")
```

Output:

```
MethodError: Cannot `convert` an object of type Int64 to an object of type String
Closest candidates are:
  convert(::Type{String}, ::String) at essentials.jl:210
  convert(::Type{T}, ::T) where T<:AbstractString at strings/basic.jl:231
  convert(::Type{T}, ::AbstractString) where T<:AbstractString at strings/basic.jl:232
  ...
```

```
Stacktrace:
```

```
[1] Robot(name::Int64, model_number::String)
```

```
@ Main ./In[25]:2
```

```
[2] top-level scope
```

```
@ In[26]:1
```

```
[3] eval
```

```
@ ./boot.jl:360 [inlined]
```

```
[4] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)
```

```
@ Base ./loading.jl:1094
```

If we give the right data types, it would work:

```
r = Robot("Zigor", 7824)
```

Output:

```
Robot("Zigor", 7824)
```

32.3. Complex Data Types

It's not that we need to have only simple data types associated with struct's attributes, we can have complex data types too. Imagine you are designing a game where there is a sprite, and its position on the screen is defined by a complex data type called position, you can define them as shown:

```
struct Position
    x::Int
    y::Int
end

mutable struct Sprite
    name::String
    position::Position
end
```

Note that `Sprite` is mutable because its position would change. I nthe code below we define a sprite with a postion

```
position = Position(5, 5)
sprite = Sprite("Turtle", position)
```

Output:

```
Sprite("Turtle", Position(5, 5))
```

We write a function called move right in which we take a `Sprite` as input, and we assign new position to it as shown

```
function move_right(sprite::Sprite)
    new_x = sprite.position.x + 1
    y = sprite.position.y

    sprite.position = Position(new_x, y)
end
```

Output:

```
move_right (generic function with 1 method)
```

Now le's move our `sprite` right

```
move_right(sprite)
```

Output

```
Position(6, 5)
```

We inspect sprite to see it’s position

```
sprite
```

Output:

```
Sprite("Turtle", Position(6, 5))
```

Its position has changed. Yaay!

Chapter 33. Vectors & Matrix

Chapter 34. Files

34.1. Plain Text

34.2. CSV

34.3. JSON

34.4. Text Vs Binary

Chapter 35. Scrapping

Chapter 36. Plots

Get the Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/plots.ipynb

Video lecture for this section could be found here https://youtu.be/BWTyQUDRXwI

36.1. Installing Julia Plots

It is said that two third of our brains is devoted to process vision. We humans are very visual animals, we can detect patters from seeing things rather than looking at a table of numbers. In fact many Data Scientists make their living by creating just visualizations and info news so that people concerned can easily digest it and understand it. I don't think I will be going to very detail about visualizations, the path for these blogs are not decided yet, but let's here look at plotting in Julia.

There is a package called Plots (https://docs.juliaplots.org/latest/) in Julia which you can install it as follows:

First launch the Julia REPL by typing in `julia` in your terminal.

```
$ julia
               _
   _       _ _(_)_     |  Documentation: https://docs.julialang.org
  (_)     | (_) (_)    |
   _ _   _| |_  __ _   |  Type "?" for help, "]?" for Pkg help.
  | | | | | | |/ _` |  |
  | | |_| | | | (_| |  |  Version 1.5.3 (2020-11-09)
 _/ |\__'_|_|_|\__'_|  |  Official https://julialang.org/ release
|__/

julia>                   |
```

Go to packages by typing in `]` at the `julia> ` prompt, then in `pkg>` type this:

```
(@v1.5) pkg> add Plots
```

so should see lot of packages getting installed and finally `Plots` too will get installed. Now the notebook for this blog is here https://gitlab.com/data-science-with-julia/code/-/blob/master/plots.ipynb. It's not that you must install Plots from Julia REPL, you can also install it pragmatically as shown below, right from your Jupyter lab:

```
# Uncomment lines below if you want to install Plots
using Pkg
Pkg.add("Plots")
```

I have commented it out in the one you can get it from Gitlab because I have already installed it using the REPL. Okay lets tell Julia that we are using `Plots` with the following statement:

```
using Plots
```

Output:

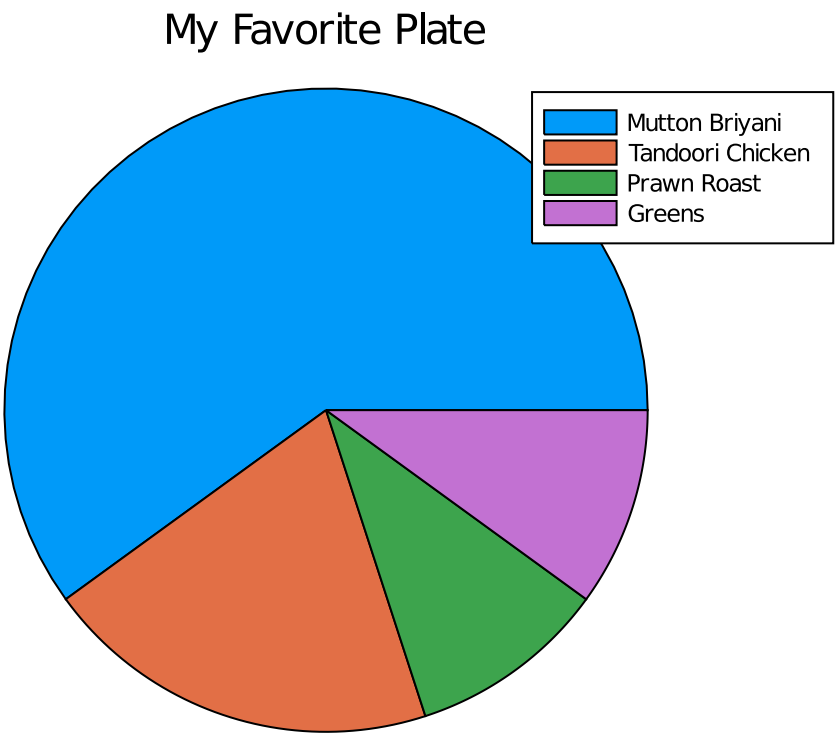
```
┌ Info: Precompiling Plots [91a5bcdd-55d7-5caf-9e0b-520d859cae80]
└ @ Base loading.jl:1278
```

This could take a while as it depends on the processing speed of your computer. For my 7-year laptop I think it took nearly 20 minutes, but once the compiling of `Plots` is doe, from the next time on its fast.

Now let's plot a simple pie chart which reflects my favorite food platter:

```
pie(
    ["Mutton Briyani", "Tandoori Chicken", "Prawn Roast", "Greens"],
    [60, 20, 10, 10],
    title = "My Favorite Plate"
)
```

Output:



We will see more of `Plots` in my upcoming sections.

36.2. Basic plot Function - Plotting Sin and Cos

To start with let's define `θ` to be from `-2π` to `2π` with steps of `0.01` as shown below:

```
θ = -2π : 0.01 : 2π
```

Output:

```
-6.283185307179586:0.01:6.276814692820414
```

Now lets plot `cos` and `sin` values of `θ`

```
plot(
    θ, [sin.(θ), cos.(θ)],
    title = "sin and cos plots",
    xlabel = "Radians",
    ylabel = "Amplitude",
    label = ["sin(θ)" "cos(θ)"],
    ylims = (-1.5, 1.5),
    xlims = (-7, 7)
)
```

Output:

images::plots/plots_2/output_4_0.svg[]

So, what just happened? We have a function called `plot()`:

```
plot()
```

To that we pass the first argument to be `θ`:

```
plot(θ)
```

the second argument to be an array of `sin` and `cos` of `θ`:

```
plot(θ, [sin.(θ), cos.(θ)])
```

apart from that we pass the following attributes too:

- `title` : Self explanatory
- `xlabel` : The text that must appear for the x-axis
- `ylabel` : The text that must appear for the y-axis

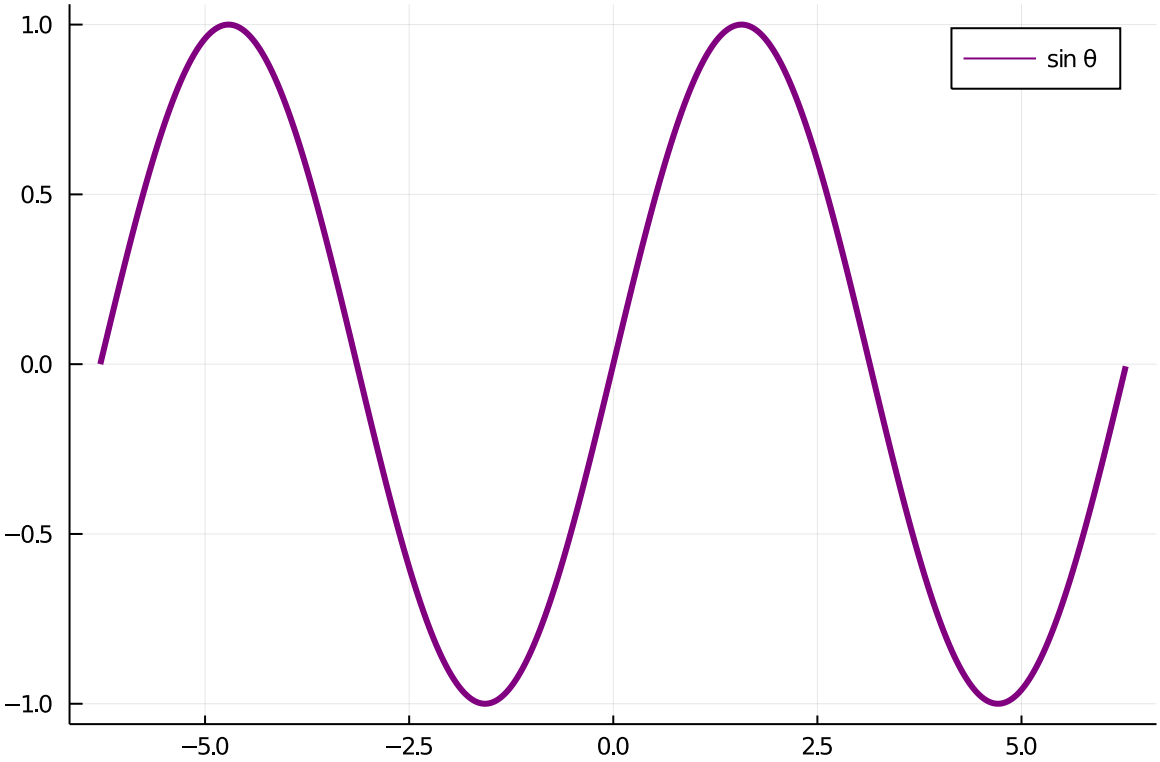
- `label` : The legend of the curves shown in top right
- `xlims` and `ylims` : The minimum and maximum of coordinates that needs to be displayed for the plot, these accepts `Tuple` as argument

36.2.1. Building a Plot

Its not that you have to create a plot all in a go, you can build it part by part, you can see blow that we plot sin`of `θ below, we have set `linewidth = 3` so it appears nice and thick, we have set the color to be purple, and its label to be sin θ:

```
p = plot(θ, sin.(θ), color = "purple", linewidth = 3, label = "sin θ")
```

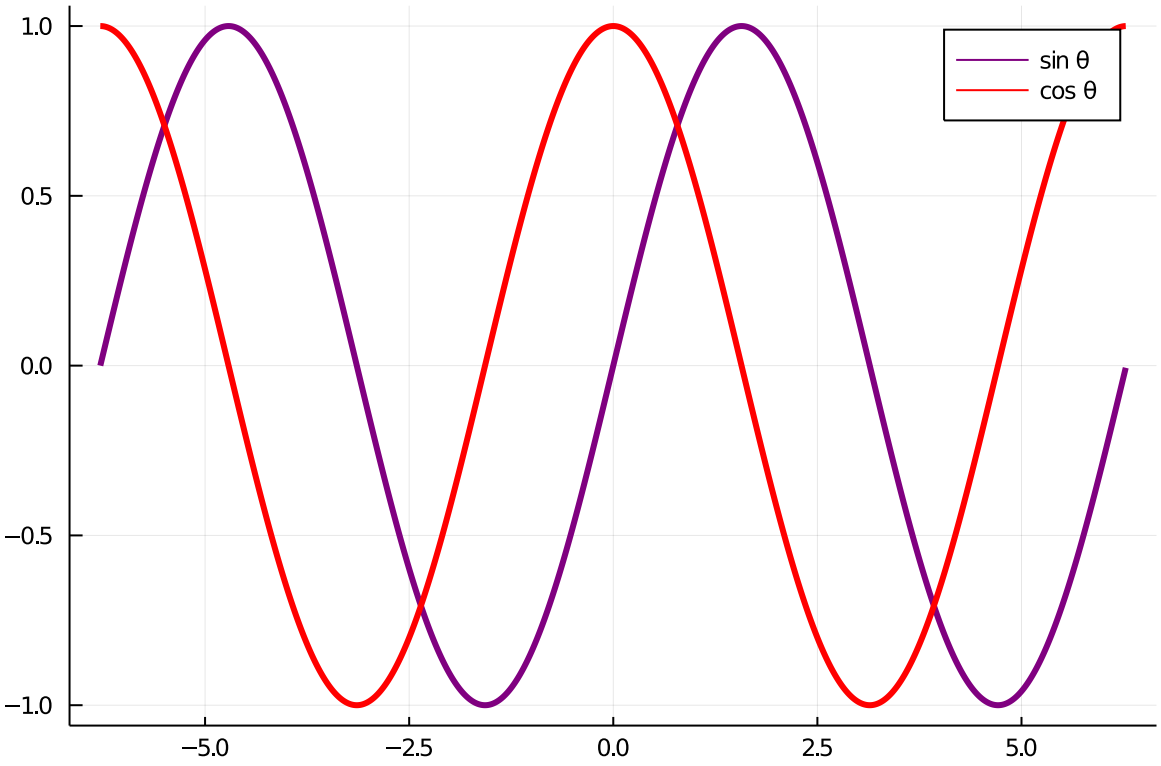
so you get a plot as shown:



we have assigned the output of the above plot to a variable named `p`, so in the future this could be modified. Now lets add `cos` plot to `p`, for that look at the code below:

```
plot!(p, θ, cos.(θ), color = "red", linewidth = 3, label = "cos θ")
```

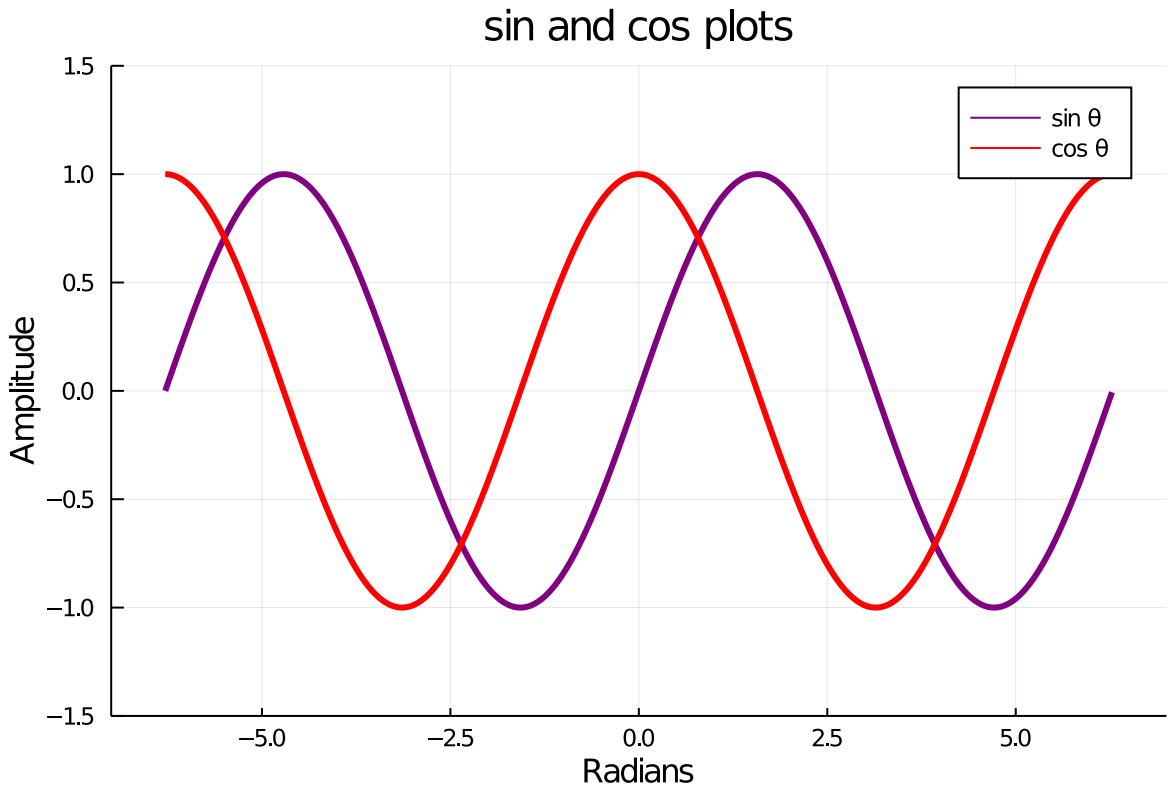
Output:



so it is the same with just one difference, we are mutating `p` by passing it to the `plot!()` function. Notice the exclamation mark `!` in the `plot!()` and it accepts a plot as its first argument which it changes. Now in the code below we further decorate `p` with title, labels and limits:

```
plot!(
    p,
    title = "sin and cos plots",
    xlabel = "Radians",
    ylabel = "Amplitude",
    ylims = (-1.5, 1.5),
    xlims = (-7, 7)
)
```

Output:

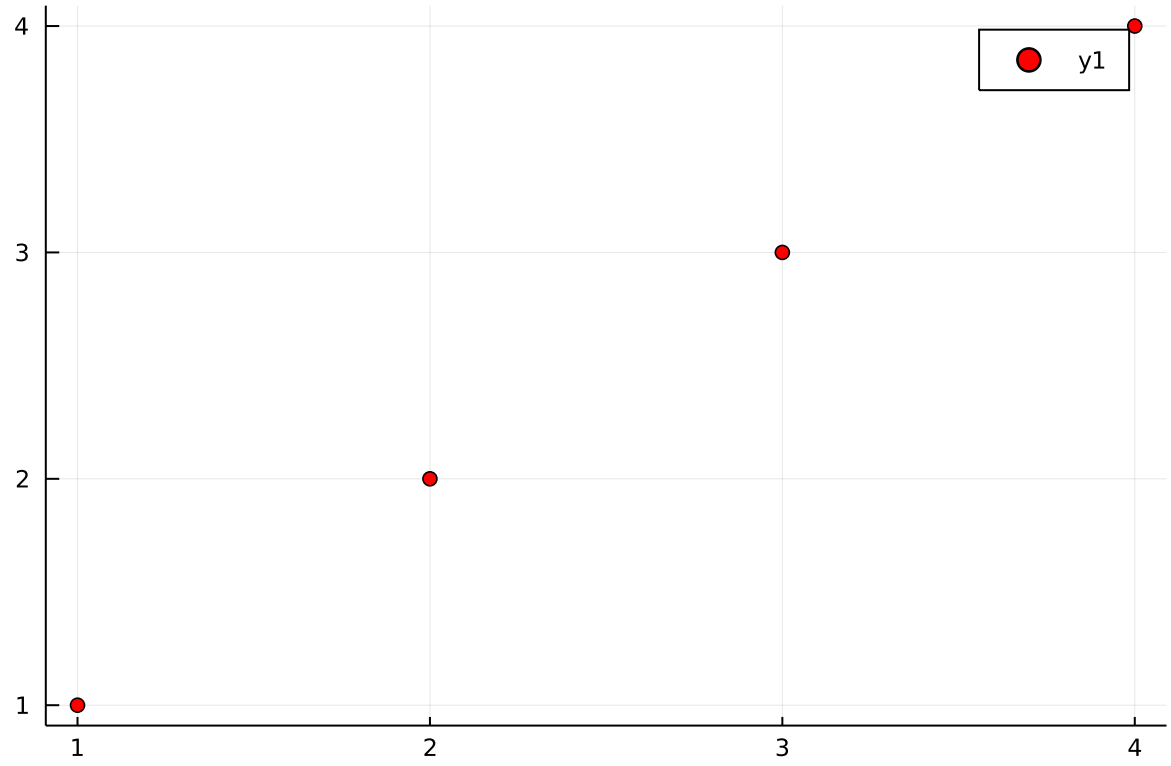


36.3. Scatter and Histogram

So let's create a simple scatter, x-values as first argument, y-values as second, I have given color as red but its completely optional:

```
scatter_plot = scatter([1, 2 , 3,  4], [1, 2, 3, 4], color = "red")
```

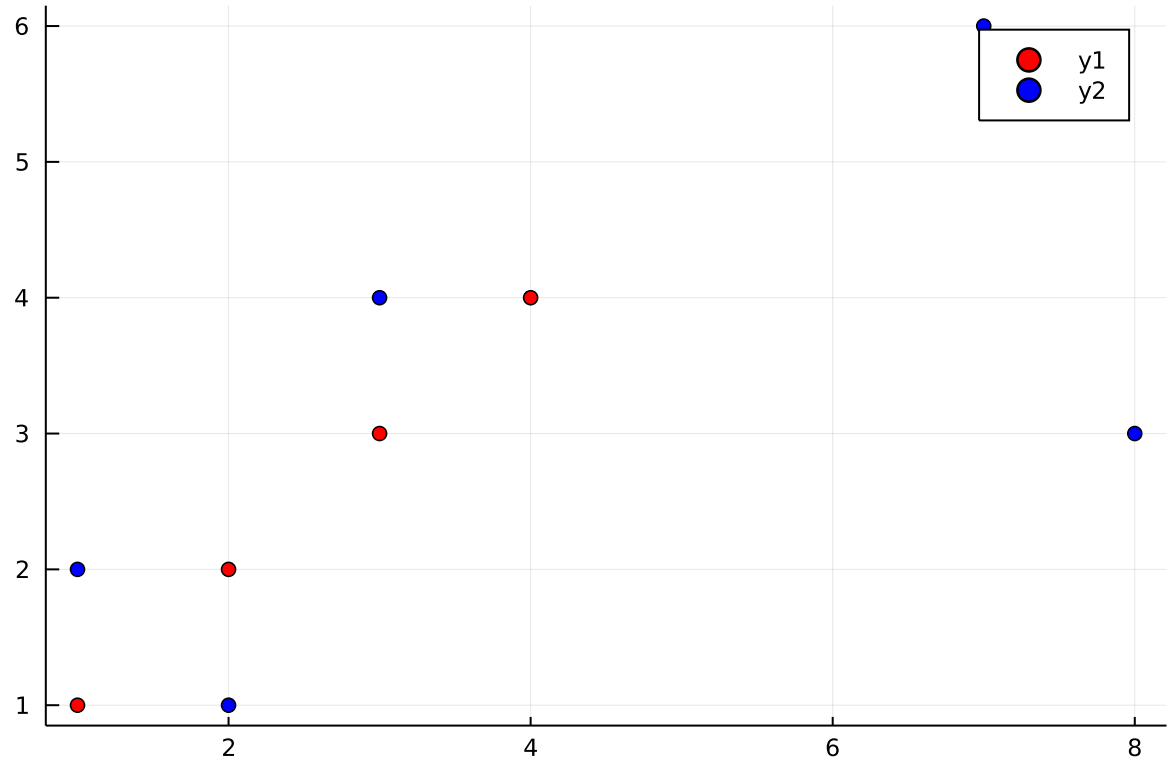
Output:



We have captured last output in a variable called `scatter_plot`, this is much better variable name I tell you. In Basic plots Functions I had given a names like `p` which is actually blasphemy in programming world, you might be smart today and understand what `p` is, but possibly 6 months later it will haunt you, or a programmer touching your code sometime later might curse you thus causing your loved ones and you to vomit blood and die. Anyway, we now add another scatter to `scatter_plot` using the code below:

```
scatter!(scatter_plot, [1, 7, 8, 2, 3], [2, 6, 3, 1, 4], color = "blue")
```

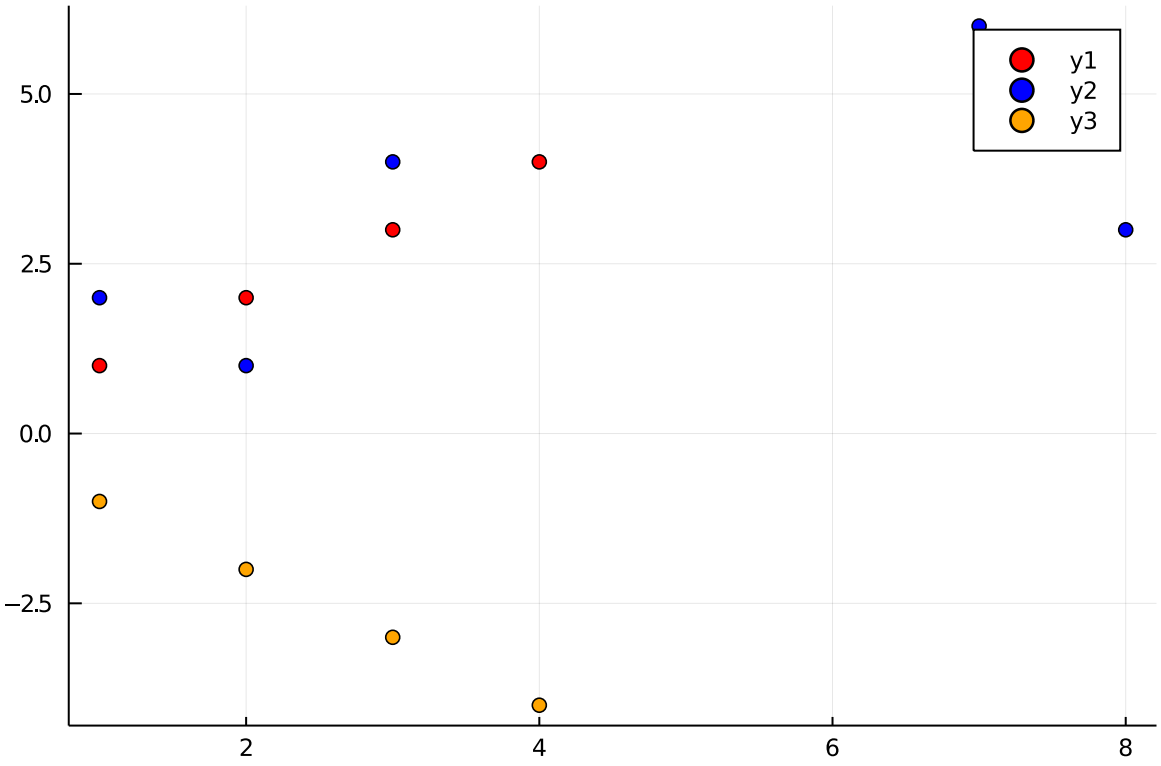
Output:



As you can see above, this time we have not used `plot!()` to modify `scatter_plot` but we have used this function `scatter!()` which I personally feel is much better to read. And we have put these new dots in color blue. Let's modify `scatter_plot` again this time using `plot!()` function as shown below:

```
plot!(
  scatter_plot,
  [1, 2, 3, 4 ],
  [-1, -2 , -3, -4],
  color = "orange",
  seriestype = :scatter
)
```

Output:



As you see above in the above `plot!()` we have used named argument `seriestype` and have set it to `:scatter` for it to be a scatter plot. I am unsure why we have a colon `:` before scatter, should check Julia docs about it. Okay, looks like this is a special kind of thing known as `Symbol`, another data type in Julia, possibly it occupies less space compared to `"scatter"` which is a string when called multiple times. I just checked this code:

```
julia> :a
:a

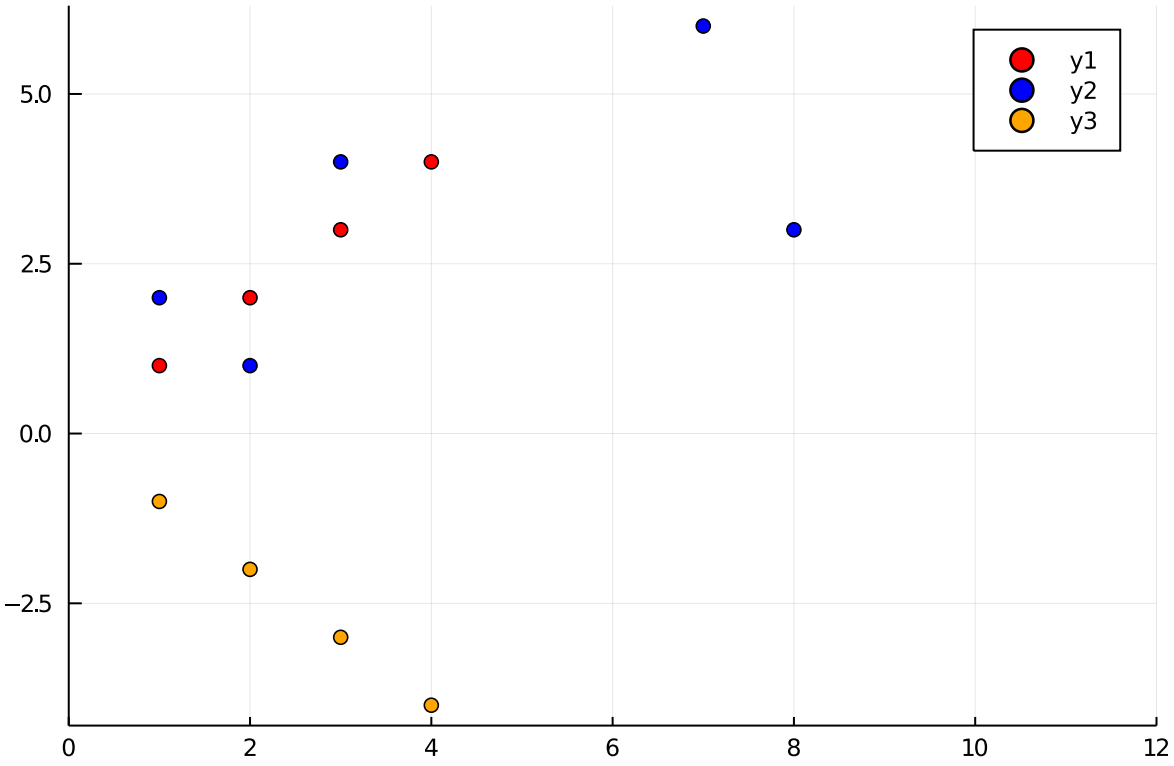
julia> typeof(:a)
Symbol
```

just to check its type.

Okay, the label box int above scatter plots is overlapping a data point, lets increase the x right limit to 12 so that it would look better:

```
plot!(scatter_plot, xlims = (0, 12))
```

Output:

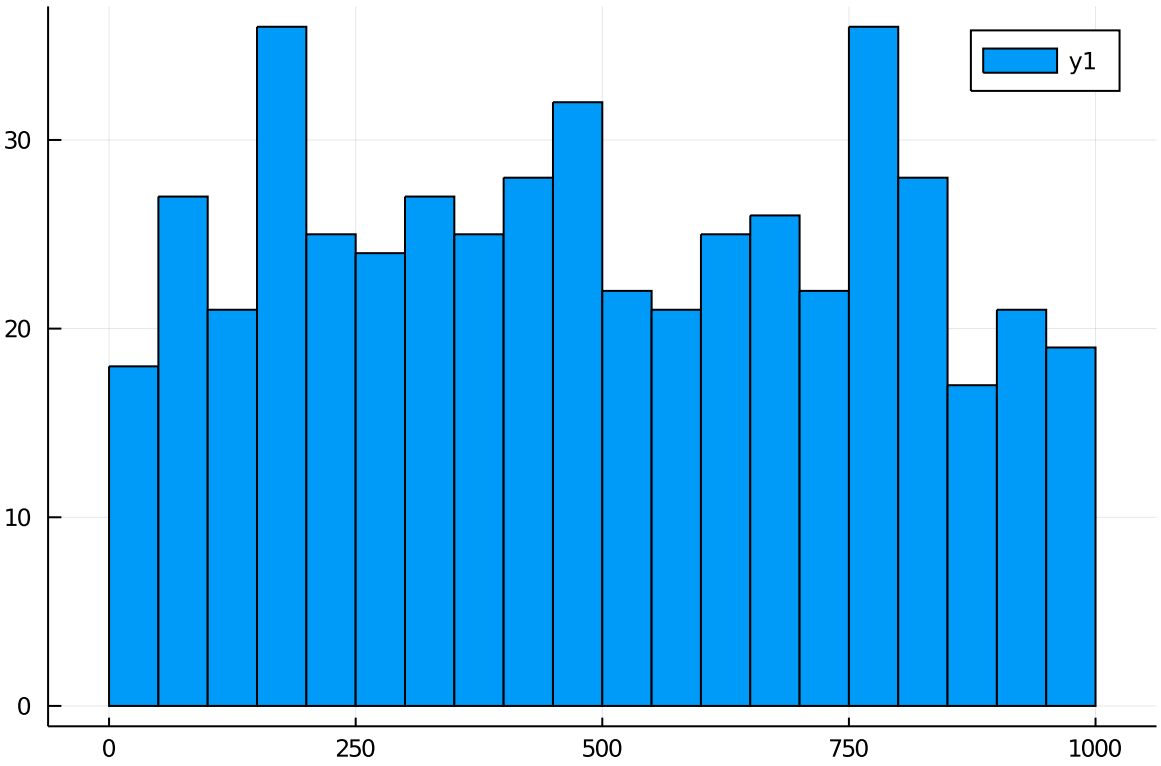


Better now!

For some reason I like histograms, so I have plotter a histogram below:

```
histogram(rand(1:1000, 500), bins = 20)
```

Output:



I think I will be using it while writing about the [Iris data set](https://en.wikipedia.org/wiki/Iris_flower_data_set). You can get the notebook file for this blog here https://gitlab.com/data-science-with-julia/code/-/blob/master/plots.ipynb.

36.4. Learn more about Plots

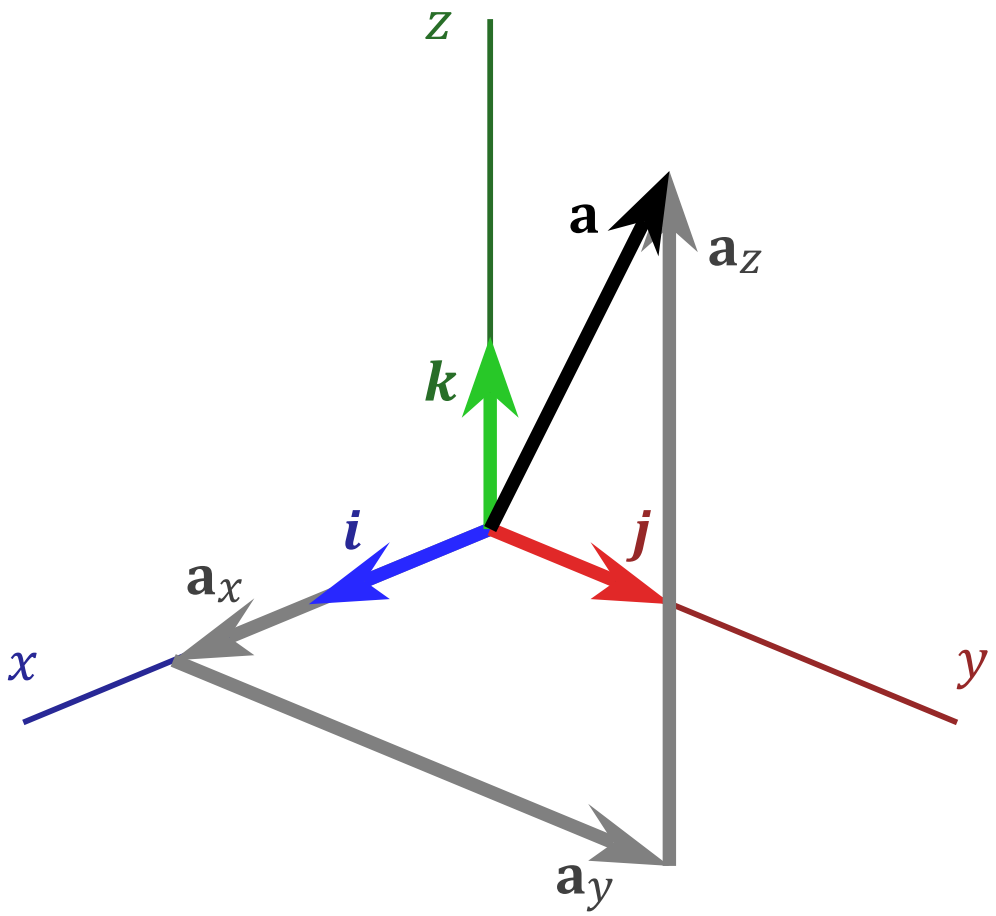
This is just a very brief section about plots and definetly you need to learn more. Please check the official Plots website https://docs.juliaplots.org/latest/, and this Julia Plots by Prude University https://www.math.purdue.edu/~allen450/Plotting-Tutorial.html is good too.

Chapter 37. Dataframes

Chapter 38. Debugging

Mathematics

Chapter 39. Vectors



Video lecture for this section could be found here https://youtu.be/468U1e4ITjI?list=PLe1T0uBrDrfOLQlomF_4AxHa4LX0wsCXa

Jupyter notebook for this section could be found here https://gitlab.com/datascience-book/code/-/blob/master/vectors.ipynb

To learn in detail about vectors, visit https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces

An array in Julia is vector. but I found some more functions needs to be written so that we can do operations like mathematical vectors. So this chapter was born. May be there is some package out there that does this, but since the aim of this book is to give me deep understanding of Data Science from its mathematical concepts, I decided to write my own functions.

39.1. Addition

So let's begin. First let's define two vectors `a` and `b` of equal length and add it.

```
a = [1, 2, 3]
b = [4, 5, 6]
a + b
```

Output:

```
3-element Vector{Int64}:
 5
 7
 9
```

This addition won't be possible if `a` and `b` are of different lengths.

39.2. reduce function

We can do the same operation explained above with `reduce` function. Say we have an array of vectors, we avoid writing loop and add them all by passing `+` sign to reduce as shown.

```
reduce(+, [a, b])
```

Output:

```
3-element Vector{Int64}:
 5
 7
 9
```

To show more the power of `reduce`, I define two more vectors `c` and `d` and add them.

```
c = [7, 8, 9]
d = [10, 11, 12]
```

Output:

```
3-element Vector{Int64}:
 10
 11
 12
```

```
reduce(+, [a, b, c, d])
```

Output:

```
3-element Vector{Int64}:
 22
 26
 30
```

39.3. Midpoint

For clustering algorithms, it's important to find mid point of vectors (which represent a point in our case). Mid point of n vectors are found out using this formula.

$\vec{m} = \frac{1}{n}\sum_{i=1}^{n} \vec{x}_i$

One should not get confused. Here $\vec{x}_i$ means it represents a point in space. In pure mathematical terms, it makes no sense to find a a mid of of n vectors.

Where $\vec{m}$ is the mid point.

So we write and midpoint function:

```
function midpoint(array_of_vectors::Vector)
    reduce(+, array_of_vectors) / length(array_of_vectors)
end
```

We test it out:

```
midpoint([a, b])
```

Output:

```
3-element Vector{Float64}:
 2.5
 3.5
 4.5
```

```
midpoint([[0, 0], [2, 2]])
```

Output:

```
2-element Vector{Float64}:
 1.0
 1.0
```

Seems to work!

39.4. Distance

We can treat a vector / array in Julia as a point and find distance beween them. Let there be two points X and Y. where we can define X as

$$X = (x_1,\ x_2,\ \text{ellipsis}\ ,\ x_n)$$

Similarly we can define Y as

$$Y = (y_1,\ y_2,\ \text{ellipsis}\ .,\ y_n)$$

Then the distance can be written as:

$$d(X,\ Y) = \sqrt{\sum_{i\ =\ 1}^{n} (x_i - y_i)^2}$$

now let's define two vectors in Julia

```
first_vector = [0, 0]
second_vector = [3, 4]
```

Now the difference between these two vectors can be written as $(x_i - y_i)$, this in Julia translates as shown:

```
difference = first_vector - second_vector
```

Output

```
2-element Vector{Int64}:
 -3
 -4
```

Now let's find $(x_i - y_i)^2$

```
difference .^ 2
```

Output:

```
2-element Vector{Int64}:
  9
 16
```

Let's sum the squares up $\sum_{i\ =\ 1}^{n} (x_i - y_i)^2$

```
sum_of_squares = sum(difference .^ 2)
```

Output:

```
25
```

Now let's take the square root to find the distance $\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$

```
sqrt(sum_of_squares)
```

Output

```
5.0
```

Let's ack up what we have done into a function:

```
function distance(first_vector::Vector, second_vector::Vector)
    Δ = first_vector - second_vector
    sum_of_squares = sum(Δ .^ 2)
    sqrt(sum_of_squares)
end
```

Let's test it out:

```
distance([0, 0], [3, 4])
```

Output:

```
5.0
```

Seems to work!

39.5. Magnitude

Given a vector $\vec{X} = (x_1, x_2, \text{ellipsis}, x_n)$, we know its magnitude can be written as:

$$\left|\vec{X}\right| = \sqrt{\sum_{i=1}^{n} x_i^2}$$

So let's translate into Julia and write a magnitude function as shown:

```
function magnitude(vector::Vector)
    sqrt(sum(vector .^ 2))
end
```

Let's test it out:

```
magnitude([3, 4])
```

Output:

```
5.0
```

Seems to work!

39.6. Unit Vector

Given a vector $\vec{X}$, we can find unit vector $\hat{X}$ using this formula:

hat X = vec X / |vec X|

where $\hat{X}$ represents the unit vector and |vec X| represents the magnitude. So now let's write this in Julia:

```
function unit(vector::Vector)
    vector / magnitude(vector)
end
```

Let's test it out:

```
unit([3, 4])
```

Output:

```
2-element Vector{Float64}:
 0.6
 0.8
```

Seems to work!

39.7. The Vector Library

I have collected the functions to operate on vectors as a library, one can get it here https://gitlab.com/datascience-book/code/-/blob/master/vectors.ipynb

We will be using this later in this book.

Chapter 40. Matrices

Video lecture for this section could be found here https://youtu.be/fYE5uUKglcU?list=PLe1T0uBrDrfOLQlomF_4AxHa4LX0wsCXa

Get Jupter note book for this section here https://gitlab.com/datascience-book/code/-/blob/master/matrix.ipynb

Let's look at Matrices with Julia in this section. To learn about Matrices thoroughly I would suggest one to visit https://www.khanacademy.org/math/precalculus/x9e81a4f98389efdf:matrices

So let's say we have a matrix A, this is how we write it mathematically:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

And this is how we represent in Julia

```
A = [1 2  3; 4 5 6]
```

Look how we have rows separated by a semicolon `;`, the columns in a row separated by space. This is what you will het as output in Jupyter notebook:

```
2x3 Matrix{Int64}:
 1  2  3
 4  5  6
```

We can ask Julia what is the type of `A` and we get is a `Matrix`

```
typeof(A)
```

Output:

```
Matrix{Int64} (alias for Array{Int64, 2})
```

`Matrix` is a in built datatype in Julia and hence we do not need to build special libraries for it.

We have A defines, now transpose of A is represented as A^T, we know this is A^T:

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

In Julia we use apostrophe to find transpose of `Matrix`, and hence A^T in Julia is written as shown:

```
A'
```

Output:

```
3x2 adjoint(::Matrix{Int64}) with eltype Int64:
 1  4
 2  5
 3  6
```

Now let's define a `Matrix` B

```
B = [7 8 9; 10 11 12]
```

Output:

```
2x3 Matrix{Int64}:
  7   8   9
 10  11  12
```

Now I can take the sum of A and B as shown:

```
A + B
```

Output

```
2x3 Matrix{Int64}:
  8  10  12
 14  16  18
```

Since `A` and `B` have the same dimensions, they add up. Similarly we can subtract Matrices as shown:

```
A - B
```

Output

```
2x3 Matrix{Int64}:
 -6  -6  -6
 -6  -6  -6
```

Now let's try to multiply `A` and `B`, and it fails

```
A * B
```

Output

```
    DimensionMismatch("matrix A has dimensions (2,3), matrix B has dimensions (2,3)")



    Stacktrace:

     [1] _generic_matmatmul!(C::Matrix{Int64}, tA::Char, tB::Char, A::Matrix{Int64},
B::Matrix{Int64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})

       @ LinearAlgebra
/Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.6/LinearA
lgebra/src/matmul.jl:814

     [2] generic_matmatmul!(C::Matrix{Int64}, tA::Char, tB::Char, A::Matrix{Int64},
B::Matrix{Int64}, _add::LinearAlgebra.MulAddMul{true, true, Bool, Bool})

       @ LinearAlgebra
/Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.6/LinearA
lgebra/src/matmul.jl:802

     [3] mul!

       @
/Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.6/LinearA
lgebra/src/matmul.jl:302 [inlined]

     [4] mul!

       @
/Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.6/LinearA
lgebra/src/matmul.jl:275 [inlined]

     [5] *(A::Matrix{Int64}, B::Matrix{Int64})

       @ LinearAlgebra
/Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.6/LinearA
lgebra/src/matmul.jl:153

     [6] top-level scope

       @ In[6]:1

     [7] eval

       @ ./boot.jl:360 [inlined]

     [8] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)

       @ Base ./loading.jl:1094
```

That's because one can multiply matrix of dimension $m \times k$ with a matrix of dimension $k \times n$ which will give matrix of dimension $m \times n$. That is the number of columns in first matrix should be equal to number of rows in second matrix, only then multiplication is possible.

However we can do element wise multiplication with A and B as shown

```
A .* B
```

Ouput

```
2×3 Matrix{Int64}:
  7  16  27
 40  55  72
```

Note that we have used `.*` operator, where the `.` means element wise and * means multiplication.

A is 2×3 matrix, let's define a matrix C which 3×2:

```
C = [1 2; 3 4; 5 6]
```

Output

```
3×2 Matrix{Int64}:
 1  2
 3  4
 5  6
```

Now we multiply both

```
A * C
```

Output

```
2×2 Matrix{Int64}:
 22  28
 49  64
```

And so above we get a 2×2 matrix.

If we multiply C and A which are of dimensions 3×2 and 2×3, we get a 3×3 matrix as shown below:

```
C * A
```

Output

```
3x3 Matrix{Int64}:
  9  12  15
 19  26  33
 29  40  51
```

Let's assign the product of `C` and `A` to a variable `D`

```
D = C * A
```

Output

```
3x3 Matrix{Int64}:
  9  12  15
 19  26  33
 29  40  51
```

Now we find inverse of `D` and assign it to a variable `D_inv`

```
D_inv = inv(D)
```

Output

```
3x3 Matrix{Float64}:
  7.03687e13  -1.40737e14   7.03687e13
 -1.40737e14   2.81475e14  -1.40737e14
  7.03687e13  -1.40737e14   7.03687e13
```

Now we multiply `D` and `D_inv`, we expect to get 3×3 identity matrix:

```
D * D_inv
```

Output

```
3x3 Matrix{Float64}:
 1.375  -0.25  0.0
 1.0     0.0   0.5
 1.0    -1.0   1.5
```

But we don't, thats because determinant of `D` is zero. We have to use a package called `LinearAlgebra` to get access to function `det` with which we can find the determinant as shown:

```
using LinearAlgebra

det(D)
```

Output

```
8.526512829121201e-14
```

In the below few examples, we find determinant of 2×2 matrix E

```
E = [1 2; 3  4]
```

Output

```
2×2 Matrix{Int64}:
 1  2
 3  4
```

```
det(E)
```

Output

```
2.0
```

and it's non zero.

Since determinant of E is non zero, we can multiply E with its invese and get an identity matrix as shown:

```
E * inv(E)
```

Output

```
2×2 Matrix{Float64}:
 1.0          0.0
 8.88178e-16  1.0
```

We know that we have defined matrix A which is a 2×3 matrix as shown:

```
A
```

Output

```
2x3 Matrix{Int64}:
 1  2  3
 4  5  6
```

Now let's find inverse of `A`

```
inv(A)
```

Output

```
    DimensionMismatch("matrix is not square: dimensions are (2, 3)")



    Stacktrace:

     [1] checksquare

       @
/Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.6/LinearA
lgebra/src/LinearAlgebra.jl:223 [inlined]

     [2] inv(A::Matrix{Int64})

       @ LinearAlgebra
/Users/julia/buildbot/worker/package_macos64/build/usr/share/julia/stdlib/v1.6/LinearA
lgebra/src/dense.jl:807

     [3] top-level scope

       @ In[18]:1

     [4] eval

       @ ./boot.jl:360 [inlined]

     [5] include_string(mapexpr::typeof(REPL.softscope), mod::Module, code::String,
filename::String)

       @ Base ./loading.jl:1094
```

And it fails. IT turns out one can find inverse of $m \times n$ matrix only when $m = n$, or when the matrix is square. But there is a way around, there is a function called `pinv` which is called pseudo inverse and this can find inverse of matrix when its not square as shown:

```
pinv(A)
```

Output

```
3x2 Matrix{Float64}:
 -0.944444   0.444444
 -0.111111   0.111111
  0.722222  -0.222222
```

Now multiplying `A` with its pseudo inverse gives a 2×2 indentity matrix as shown:

```
A * pinv(A)
```

Output

```
2x2 Matrix{Float64}:
  1.0          2.22045e-16
 -8.88178e-16  1.0
```

How ever if a matrix determinant is zero, still `pinv` does not work, we multiply `D` with its pseudo inverse and we do not get identity matrix as shown

```
D * pinv(D)
```

Output

```
3x3 Matrix{Float64}:
  0.833333  0.333333  -0.166667
  0.333333  0.333333   0.333333
 -0.166667  0.333333   0.833333
```

As you can verify once again, determinant of `D` is almost zero

```
det(D)
```

Output

```
8.526512829121201e-14
```

Now let's look at matrix division. One may note that we use `/` to divide matrix. Let define a matrix and store it in variable `A`

```
A = [1 2; 3 4]
```

Output

```
2x2 Matrix{Int64}:
 1  2
 3  4
```

Similarly we define matrix `B`

```
B = [5 6; 7 8]
```

Output

```
2x2 Matrix{Int64}:
 5  6
 7  8
```

Now we can divide `A` by `B` as shown

```
A / B
```

Output

```
2x2 Matrix{Float64}:
 3.0  -2.0
 2.0  -1.0
```

This is same as `A` multiplied by inverse of `B`

```
A * inv(B)
```

Output

```
2x2 Matrix{Float64}:
 3.0  -2.0
 2.0  -1.0
```

We see `A` mutiplied by `pinv(B)` and we still get the same result.

```
A * pinv(B)
```

Output

```
2x2 Matrix{Float64}:
 3.0  -2.0
 2.0  -1.0
```

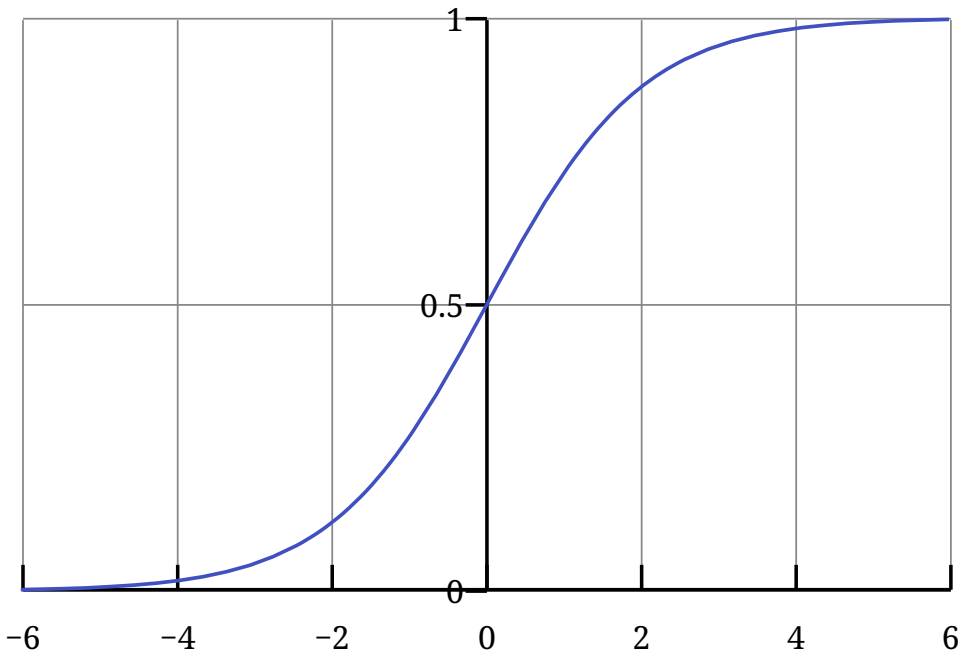
As a rule of thumb, I would suggest one to use `pinv` rahter than `inv`.

Chapter 41. Sigmoid

Video lecture for this section could be found here https://youtu.be/CKIsnEcg7NI

Get the Jupyter notebook here https://gitlab.com/datascience-book/code/-/blob/master/sigmoid.ipynb

A Sigmoid function is used to squash any values between 0 and 1, take a look at this curve



You would see its 0.5 at $x = 0$ and below it it decreases from 0.5 to 0, and above it it increases from 0.5 to 1.

Sigmoid mathematically is been represented by this formula

$S(x) = \frac{1}{1 + e^{-x}}$

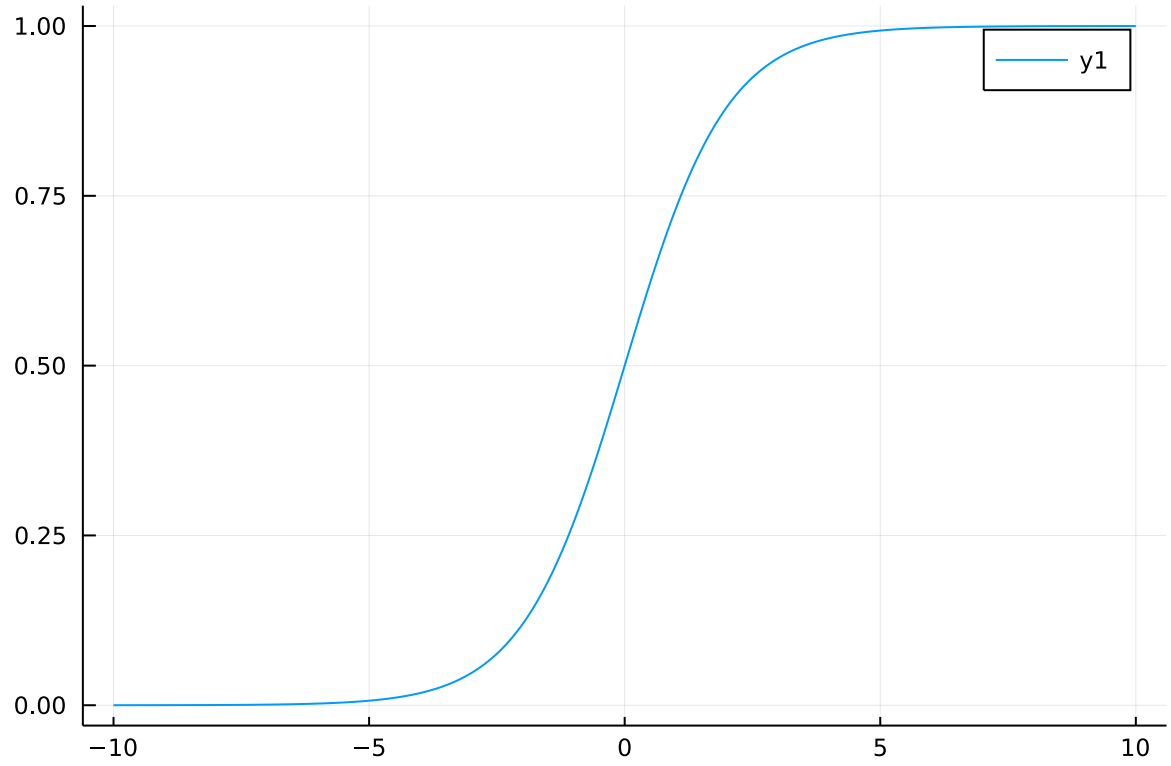
We can write a sigmoid function in Julia as shown

```
sigmoid(z) = 1 / (1 + exp(-z))
```

Now let's plot a sigmoid from $x = -10$ to $x = 10$

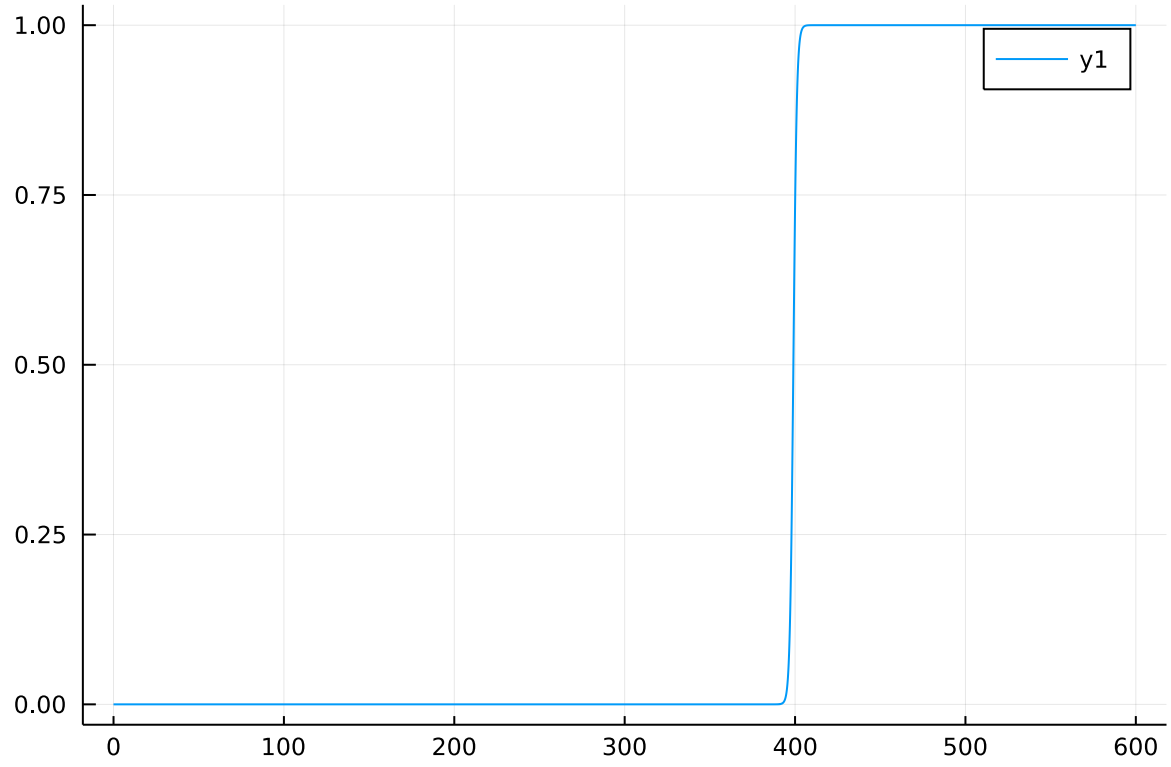
```
x = -10:0.01:10
y = sigmoid.(x)
using Plots
plot(x, y)
```

so this is how the plot looks like:



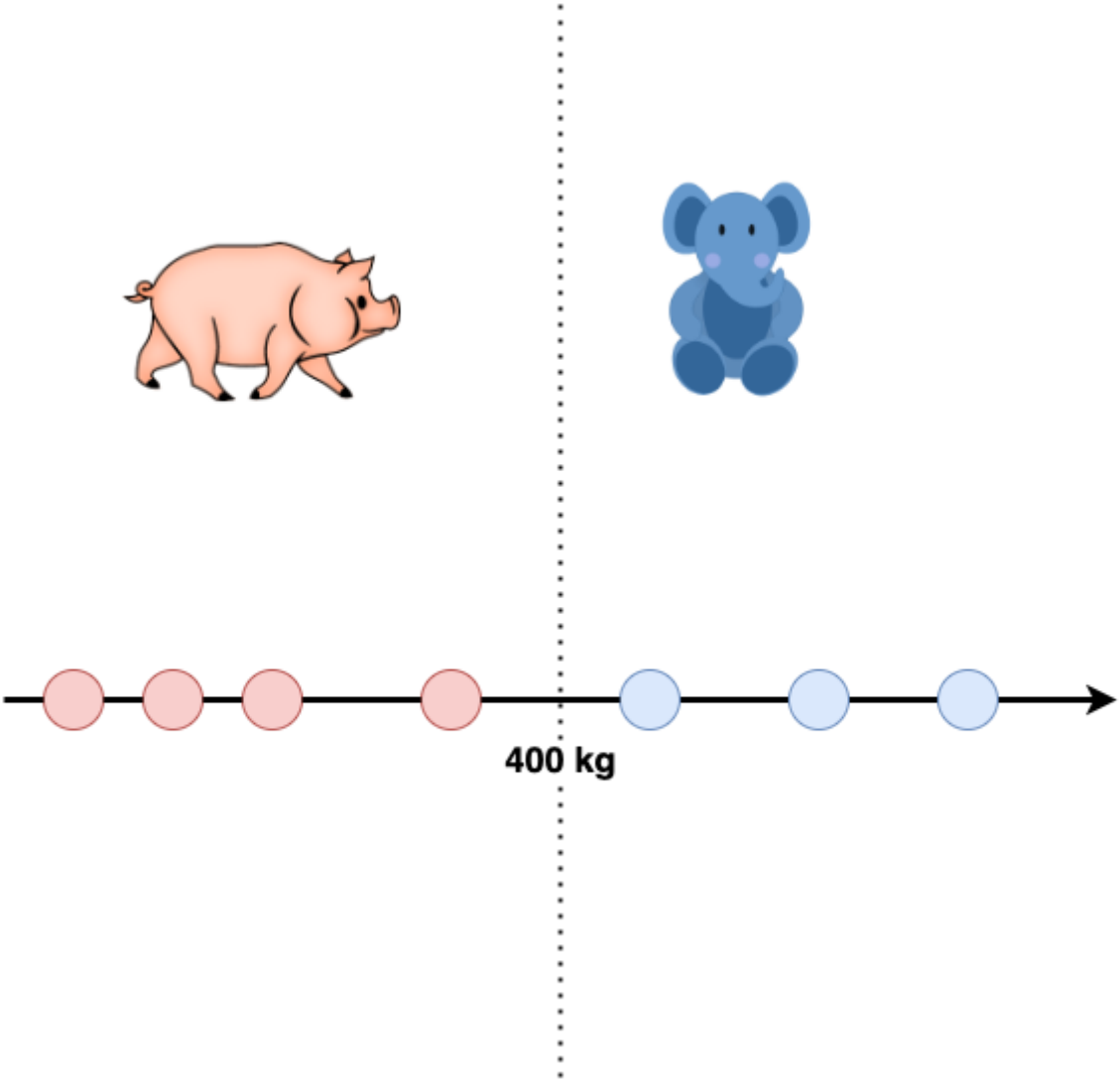
It never goes below zero and never crosses above 1.

```
x = 0:0.01:600
y = sigmoid.(x .- 399)
plot(x, y)
```

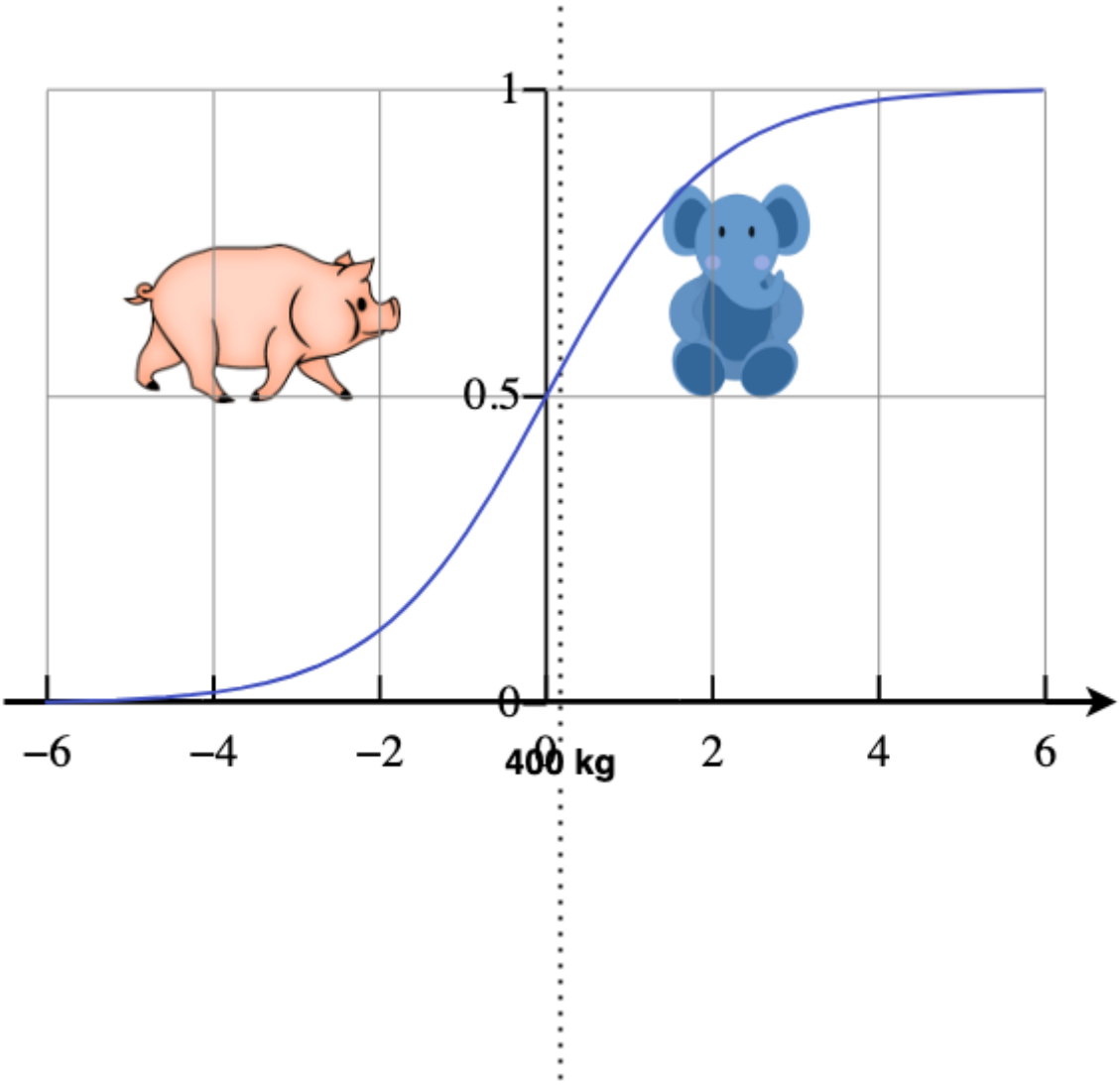


Now let's say we have data of animal weight's and its as shown. You notice that the animals that

weigh less than 400 Kg are classified as pigs and one above that are classified as baby elephants.



Now let's say we can manipulate a sigmoid graph in such a way that its $y = 0.5$ occurs slightly before $x = 400$ as shown below:



then we can say we have written a classification equation which outputs values less than or equal to 0.5 for pigs and above 0.5 for baby elephants. Turns out that we can write such a classifier as shown:

```
function classify_animal(weight)
    value = sigmoid(weight - 399)

    if value > 0.5
        "Baby Elephant"
    else
        "Pig"
    end
end
```

And it works:

```
classify_animal(350)
```

Output:

```
"Pig"
```

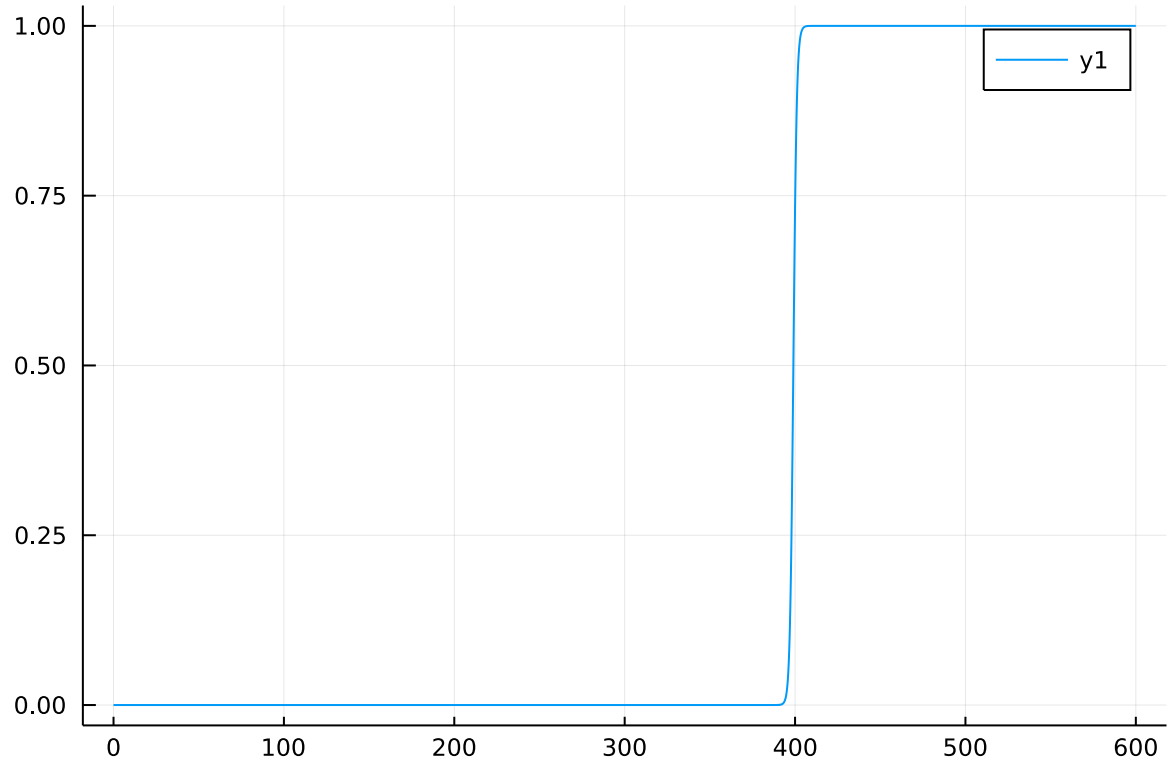
```
classify_animal(450)
```

Output:

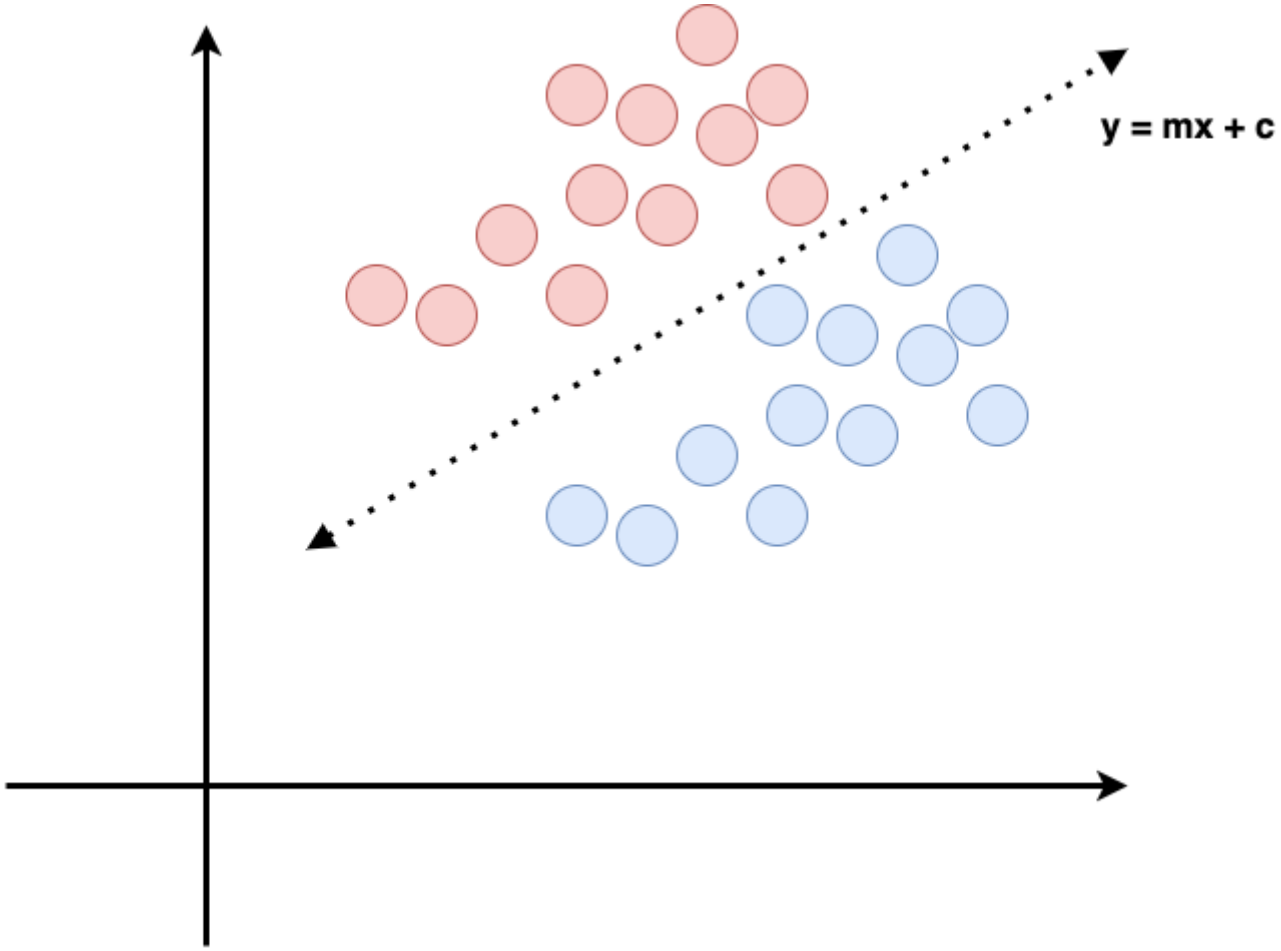
```
"Baby Elephant"
```

Or to represent it more linearly, we can pass equation of line $mx + c$ to the sigmoid function where $m = 1$ and $c = -399$ which does the same thing of giving 0.5 or lower for pigs and above 0.5 to 1 for baby elephants.

```
x = 0:0.01:600
m = 1
c = -399
y = sigmoid.(m * x .+ c)
plot(x, y)
```



Now this is just classification for points in a line. As an exercise, think about 2-D classification as shown below, where you need to classify bunch of points below a line as blue, and above that point as red. How will you do it?



Hint: equation of plane is $z = ax + by + c$, now think of $S(z)$, that is `sigmoid.(z)`, the one above 0.5 will be called as pink and one below it will be called as blue.

Chapter 42. Bayesian

Chapter 43. Statistics

Get the Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/stats.ipynb

Video lecture for this section could be found here https://youtu.be/72DfAh5qraU and here https://youtu.be/G2gB1BXPGr4

Data Science is all bout numbers, so let' do statistics on a bunch of numbers. We won't be diving deep into statistics, but tat the end of this section I will be giving you some reference sections where you can learn more.

There are two packages that can be used for stats in Julia, they are LinearAlgebra (https://docs.julialang.org/en/v1/stdlib/LinearAlgebra/) and Stats (https://docs.julialang.org/en/v1/stdlib/Statistics/). There is also a comprehensive StatsKit (https://juliastats.org/). But here we like to do things from scratch and hence we will build our own stats library and call it `stats_lib.jl`.

In future we might use it in other parts of this book. So let's dive into statistics.

43.1. Total

One of the basic operations of statistics is totaling an array, it's been represented by the following formula, where the Σ represents summation.

$$\text{total} = \sum_{i=1}^{n} x_i$$

the equation above means that, presented with an array of length `n`, we add all values up. So it can be expanded as:

$$\sum_{i=1}^{n} x_i = x_1 + x_2 + \text{ellipsis} .. + x_n$$

In Julia we write our own sum function like this

```
function sum(v::Vector)
    total = 0

    for element in v
        total += element
    end

    total
end
```

And let's test it out

```
sum([1, 2, 3])
```

Output:

```
6
```

43.2. Minimum

There is a function called minimum in julia which can be used like this

```
minimum([1, 2, 3])
```

But then since we are bad ass programmers, let's write our own minimum as follows

```
function vect_min(v::Vector)
    min = v[1]

    for element in v
        if element < min
            min = element
        end
    end

    return min
end
```

Let's test it out:

```
vect_min([6, 5, -1])
```

Output

```
-1
```

I hope the reader of the book does not need an explanation what minimum is.

43.3. Maximum

Just like minimum, let's write our own maximum function, though Julia provides a `maximum` function which can be used like this:

```
maximum([1, 2, 3])
```

```
function vect_max(v::Vector)
    max = v[1]

    for element in v
        if element > max
            max = element
        end
    end

    return max
end
```

```
vect_max([6, 5, -1])
```

Output

```
6
```

Once again, I hope the reader knows what an maximum is.

43.4. Range

Range is the maximum spread of our data, it is nothing bit minimum minus the maximum of values in a vector, so we can define our range. There is a `range` function in Julia, but it does a different thing, so we write down our own range function as shown:

```
function vect_range(v::Vector)
    vect_max(v) - vect_min(v)
end
```

And we test it out:

```
vect_range([7, 1, 5, 2])
```

Output

```
6
```

43.5. Mean

Mean is average of all values. Usually statisticians talk about two means, let's say you have entire population of India, and you want to know whats the average earning of a person, you can't ask

every person in a nation of 1.4 billion and average out, instead you ask a selected sample of people and average them out, this is called sample mean. If n peoples earnings are asked and averaged out, then we represent that mean as sten[bar x] (a bar above x) as shown:

$$\overline{x} = \frac{\sum_{i=1}^{n} x_i}{n}$$

Let's say you work in a ten people office, and you want to know how many plates of biriyani each person eats in a week, you can ask every one and then average it out. Since every person in study is covered here, we call this population mean and we represent it by Greek letter μ mu, and since we want to represent total sample count we represent it with capital N as shown below

$$\mu = \frac{\sum_{i=1}^{N} x_i}{N}$$

Now let's write our own function to find mean:

```
function mean(v::Vector)
    sum(v) / length(v)
end
```

Let's test it out

```
mean([1, 2, 3, 4])
```

Output:

```
2.5
```

43.6. Median

You would have heard about medians in roads, it will be in the middle of the road. A Median of a vector of values is nothing but the middle value of sorted vector.

You have 2 cases here, take this vector `[1, 2, 3, 5, 4]`, if you sort it, it will become `[1, 2, 3, 4, 5]`, the middle value is 3 which you can take it as a median. But what if you have a vector like `[1, 2, 3, 4]`, it will have two values in the middle which are `2` and `3`, in that case take the average of them both so you get $\frac{2+3}{2} = 1.5$.

So we write our own median function which goes like this:

```
function median(v::Vector)
    v = sort(v)
    n = length(v)
    middle = Int(ceil(n/2))

    if n % 2 == 0
        (v[middle] + v[middle + 1]) / 2
    else
        v[middle]
    end
end
```

Let's test it out

```
median([1, 2, 3, 5, 4])
```

Output

3

```
median([2, 1, 3, 4])
```

Output

```
2.5
```

43.7. Mean Vs Median

Let's say that it's Tina's birthday, she distributes sweets to every one in her class, there are 15 students in the class. Every student get's one of them, but Tina's close friends gets 5, 7, 10, and Tina herself eats 10. So if you put these values in a vector, you will get it as shown:

```
sweet_count = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 7, 10, 10]
```

Now let's find its mean:

```
mean(sweet_count)
```

Output

```
2.8666666666666667
```

So its nearly 2.9 sweets per person which is way off than 1, in fact its 190% off. It will not do justice if we tell most of the people ate neatly 3 sweets which is not true. In other worlds mean gets wildly upset or gives wrong figures if there are outliers. Here the outlier values are 5 and above.

Now let's take the median:

```
median(sweet_count)
```

Output

```
1
```

Surprisingly median says's most had just 1 sweet, which is true. In other words median dies not get's swayed by outliers or very large values that pop into an array containing lots of values that crowd around some point.

43.8. Mode

In a given set of observations, what ever values repeats the most, that one is taken as mode. Take the example below where the number of sweets distributed to a class of students by a student during her birthday is given:

```
sweet_count = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 7, 10, 10]
```

In the code below, we include `ml_lib.jl` where we have the counter function,

```
include("lib/ml_lib.jl")

function mode(v::Vector)
    counts = counter(v)

    counts_max_to_min = sort([(val, key) for (key, val) in counts], rev = true)
    counts_max_to_min[1][2]
end
```

and we gets the counts of values in passed `Vector v` in this line:

```
counts = counter(v)
```

If we pass `sweet_count` to counter, we might get it like:

```
[(11, 1), (1, 5), (1, 7), (2, 10)]
```

It returns a Array of Tuples, where the first element of Tuple is the count, and second is value.

In this line

```
counts_max_to_min = sort([(val, key) for (key, val) in counts], rev = true)
```

we sort the count in descending order of counts, so it becomes like this:

```
[(11, 1), (2, 10), (1, 5), (1, 7)]
```

Now we take the first tuple `counts_max_to_min[1]`, so we get `(11, 1)`, the second element of the Tuple is the value, so we return it using the following command `counts_max_to_min[1][2]`.

Now let's compute the mode of `sweet_count` array:

```
mode(sweet_count)
```

Output

```
1
```

43.9. Percentile

Let's say that you get a salary of Rs 100,000/-, and 98% of Indians earn less than you, then you are in the 98th percentile. If we say we collect a fair sample of 1000 salaries, and take the 98th percentile of it, that is the 980th element of sorted salaries,

$$1000 \times 98\% = 1000 \times \frac{98}{100} = 980$$

then the value of the element should be close to 100,000.

So let's write our own percentile function as shown:

```
function percentile(v::Vector, p)
    index = trunc(Int, p * length(v))
    sort(v)[index]
end
```

Let's test it out:

```
vector = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
percentile(vector, 0.5)
```

Output

```
5
```

```
percentile(vector, 0.75)
```

Output

```
7
```

Seems to work!

43.10. Interquartile Range (IQR)

One can learn more about IQR here https://en.wikipedia.org/wiki/Interquartile_range

Simply put, interquartile range is the difference between 25th and 75th percentile, so we can code it as shown:

```
function iqr(v::Vector)
    percentile(v, 0.75) - percentile(v, 0.25)
end
```

And let's test it out

```
vector = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
iqr(vector)
```

Output

```
5
```

43.11. Variance

Variance σ^2 is a measure of how much data points in a vector vary from its mean sigma:[mu]. Some of the data point can lie above mean and some of it can lie below the mean, this means one that are more positive and more negative than the mean should not cancel out, so in order to find the

variance we subrect the mean from the data point $x - \mu$ and square it $(x - \mu)^2$ so that it's always positive. We want find the average variation, so we divide it by number of data points, so we finally we get variance as shown:

$$\sigma^2 = \frac{\sum (x - \mu)^2}{N}$$

Let's say that you are conducting a research about equality of pay, you want to start from your office to find out how equal pay is, you can use the above formula to find it.

LEt's say that you want t find equality of pay for the entire tech industry, it's impossible to collect data from every one in the tech industry, so to compensate for the unknown, if you had collected n data points, you use the formula below:

$$s^2 = \frac{\sum (x - \bar{x})^2}{n - 1}$$

Don't ask me what the $n - 1$ is, I really don't get it fully, possibly you can refer here: https://www.khanacademy.org/math/statistics-probability/summarizing-quantitative-data#variance-standard-deviation-sample

```
function variance(v::Vector, kind = :population)
    if kind == :population
        population_variance(v)
    else
        sample_variance(v)
    end
end

function population_variance(v::Vector)
    N = length(v)
    x = v
    μ = mean(v)
    mean_difference = x .- μ
    sq_mean_diff = mean_difference .^ 2
    sum(sq_mean_diff) / N
end

function sample_variance(v::Vector)
    n = length(v)
    x = v
    average = mean(v)
    avg_difference = x .- average
    sq_avg_diff = avg_difference .^ 2
    sum(sq_avg_diff) / (n -1)
end
```

So we wrote two functions that calculates population variance which is handled by `population_variance(v::Vector)` and sample variance which is handled by `sample_variance(v::Vector)`. We wrote a generic function called variance `variance(v::Vector, kind = :population)` which has an second argument which when by default calculates population variance, or calculates it when `:population` is passed as second argument. It calculates sample

variance otherwise.

Now we test these functions out ash shown below.

```
population_variance([6, 2, 3, 1])
```

Output

```
3,5
```

```
variance([6, 2, 3, 1])
```

Output

```
3,5
```

```
variance([6, 2, 3, 1], :population)
```

Output

```
3.5
```

```
sample_variance([6, 2, 3, 1])
```

Output

```
4.666666666666667
```

```
sample_variance([2, 5, 6, 1])
```

Output

```
5.666666666666667
```

```
variance([6, 2, 3, 1], :sample)
```

Output

```
4.666666666666667
```

In the below example, we have two array's, one is amount of sweet eaten by an entire class of students which we capture it in a variance `sweet_count`, and we take a sample of it in `sample_sweet_count`.

```
sweet_count = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 7, 10, 10]
sample_sweet_count = [1, 1, 1, 1, 1, 1, 5, 10]
```

Now we find the variance of the population.

```
variance(sweet_count)
```

Output

```
10.782222222222222
```

Next we find the variance of `sample_sweet_count`.

```
variance(sample_sweet_count)
```

Output

```
9.484375
```

They naturally differ, but look below, if I do a sample variation on `sample_sweet_count`, it comes near to the population variance `10.782222222222222`.

```
variance(sample_sweet_count, :sample)
```

Output

```
10.839285714285714
```

Though this example might be bit tailored, I hope one to find out about the division by $n-1$, and convince oneself that it works through proper reasoning.

43.12. Standard Deviation

Calculating Standard Deviation https://www.khanacademy.org/math/statistics-probability/summarizing-quantitative-data/variance-standard-deviation-population/a/calculating-standard-deviation-step-by-step

Let's say we have this sweet distribution on Tina's birthday, if you look at the variance, its dimension would be sweet^2, that is we are squaring things, so it may give some sense of spread, but it would not be that intuitive.

So taking the square root will make the dimension to sweet, and so it would be bit more intuitive to know about the spread.

Square root of variance is called standard deviation. The population standard deviation is represented by σ (sigma) as shown below

$$\sigma = \sqrt{\frac{\sum (x - \mu)^2}{N}}$$

The sample standard deviation is represented by s as shown below.

$$s = \sqrt{\frac{\sum (x - \bar{x})^2}{n - 1}}$$

Having coded functions for variance, standard deviation is nothing but square root of it, so we code it as follows.

```
function standard_deviation(v::Vector, kind = :population)
    variance = if kind == :population
        population_variance(v)
    else
        sample_variance(v)
    end

    sqrt(variance)
end
```

Now let's test it out

```
sweet_count = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 5, 7, 10, 10]
sample_sweet_count = [1, 1, 1, 1, 1, 1, 5, 10]
```

```
standard_deviation(sweet_count)
```

Output

```
3.283629428273267
```

```
standard_deviation(sweet_count, :sample)
```

Output

```
3.39887936714161
```

43.13. Covariance

We have seen that Variance is a measure of spread. Covariance sees if values in two given vectors vary in the same way. That is if we hav vectors x and y, covariance checks if x_i and y_i both lie above or below their respective means $\bar{x}$ and $\bar{y}$. If both are above or if both are below their means, then covariance increases, else it decreases.

Mathematically this is the formula for covariance:

$$cov(x, y) = \sum_{i=1}^{n} \frac{(x_i - \bar{x}) \cdot (y_i - \bar{y})}{n - 1}$$

And this is how we coded it:

```
function covariance(x::Vector, y::Vector)
    x_mean = mean(x)
    y_mean = mean(y)
    x_diff_mean = x .- x_mean
    y_diff_mean = y .- y_mean

    n = length(x)
    sum(x_diff_mean .* y_diff_mean)  / (n - 1) # I have no clue about this n - 1
end
```

Now let's test it out on an array that is dissimilar

```
x = [1, 3, 2, 5, 8, 7, 12, 2, 4]
y = [8, 6, 9, 4, 3, 3,  2, 7, 7]

covariance(x, y)
```

Output

```
-8.069444444444445
```

As you see on dissimilar arrays it's low, n the above case it drops below zero. How ever on arrays where both of their values vary in some kind of synchorony, the covariance is high as shown below.

```
x = [1, 2, 4, 5]
y = [1, 3, 5, 7]

covariance(x, y)
```

Output

```
4.666666666666667
```

43.14. Correlation

Covariance could b any number, there is no lower of upper bound, plus think of their unit, if we give first array as amount of time a kid plays games and the second is marks, the covariance unit will be *hour-marks*, looks funny isn't it?

So how to mitigate this problem, wouldn't it be great if we can have a unit less number from -1 to 1, where the more minus means two arrays are not correlated at all, and as it approaches 1, it means that the arrays are really correlated.

Turns out that if you find covariance of two vectors, and divide them by standard deviation of the two vectors, then you get their correlation. So mathematically we can write correlation of two arrays x and y as shown.

$$crr(x, y) = \frac{cov(x, y)}{s_x s_y}$$

Where

- $crr(x, y)$ is the correlation of vectors x and y
- $cov(x, y)$ is the covariance of vectors x and y
- s_x is the standard deviation of x
- s_y is the standard deviation of y

We have coded correlation as follows

```
function correlation(x::Vector, y::Vector)
    covariance(x, y) / (standard_deviation(x, :sample) * standard_deviation(y, :
sample))
end
```

So for highly correlated data sets, the correlation approaches nearly 1 as shown.

```
x = [1, 2, 4, 5]
y = [1, 3, 5, 7]

correlation(x, y)
```

Output

```
0.9899494936611666
```

For data sets that are not correlated, the correlation approaches to -1 as shown.

```
x = [1, 3, 2, 5, 8, 7, 12, 2, 4]
y = [8, 6, 9, 4, 3, 3,  2, 7, 7]

correlation(x, y)
```

Output

```
-0.9069095825045426
```

43.15. Reference

1. Statistics, Khan Academy - https://www.khanacademy.org/math/statistics-probability/
2. Introductory Statistics - https://open.umn.edu/opentextbooks/textbooks/introductory-statistics
3. Online Stats Book - https://onlinestatbook.com/
4. Introductory Statistics (Openstax) https://openstax.org/details/introductory-statistics

Chapter 44. Probability

I'm not a expert in probability, but I will present what I know. Let say there is a event happening, say you are rolling a dice, and let A represent the probability of getting a 6, then mathematically the probability of A happening is represented by $P(A)$. And we all know if a dice is rolled say a 1000 times or more $P(A)$ will approach close to $\frac{1}{6}$.

44.1. Independent and Dependent Events

Let's say we are rolling dice twice, what is the probability of getting 1? We can can that $P(1) = \frac{1}{6}$. Now we roll the dice for the second time, what is the probability of getting 5 the second time? We can say it's $P(5) = \frac{1}{6}$.

Now if we ask the question, if we throw the dice twice, what is the probability of getting 1 the first time and 5 the second time? This mathematically is represented by $P(1, 5)$, which is nothing but the product of of $P(1)$ and $P(5)$.

$$P(1, 5) = P(1) \cdot P(5)$$

We can say that getting a 5 in the second roll is independent of getting 1 in the first roll. That is they are independent, no information is stored in the dice saying that I got 1 the first time, so reduce the probability of getting 1 the second time and increase the probability of getting 5 the second time. So these are called independent events.

If two events are independent, then the probability of one happening given another has happened is given simple multiplication of the two probilities as shown below:

$$P(A,\ B) = P(A) \cdot P(B)$$

44.2. Monte Carlo Simulation

44.3. Bayes Theorem

$$P(A|B) = \frac{P(A,\ B)}{P(B)}$$

$$P(B|A) = \frac{P(B,\ A)}{P(A)}$$

$$\{P(A|B)\frac{}{P(B|A)}\} = \frac{P(A,\ B)}{P(B)} \div \frac{P(B,\ A)}{P(A)}$$

$$\{P(A|B)\frac{}{P(B|A)}\} = \frac{P(A,\ B)}{P(B)} \cdot \frac{P(A)}{P(B,\ A)}$$

$$\{P(A|B)\frac{}{P(B|A)}\} = \frac{P(A,\ B)}{P(B)} \cdot \frac{P(A)}{P(A,\ B)}$$

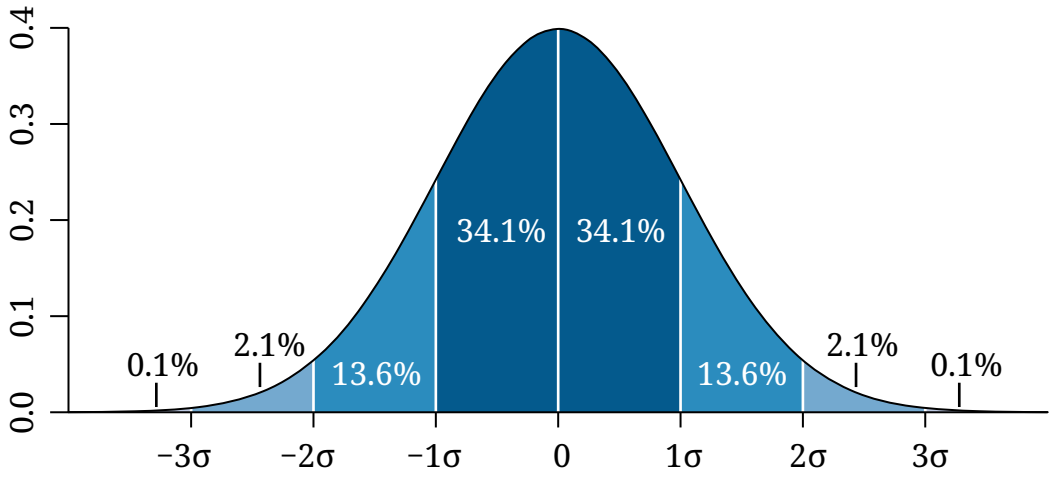
$$\{P(A|B)\frac{}{P(B|A)}\} = \frac{P(A)}{P(B)}$$

$$P(A|B) = P(B|A) \cdot \frac{P(A)}{P(B)}$$

$$P(B|A) = P(A|B) \cdot \frac{P(B)}{P(A)}$$

44.4. Normal Distribution Curve

$$f(x|\mu,\ \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \cdot \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$



Machine Learning

Chapter 45. Genetic Algorithms

45.1. Guessing a Number with Genetic Algorithm

Video lecture for this section could be found here https://youtu.be/VOrVxAOJn3o

What is genetics? What makes us change? Well let's take for instance the corona virus pandemic, there are some humans who are naturally immune to it, some people who are not. Say that we are not progressed so scientifically, then what would have happened? People who are not immune to it will perish, some will have reduced biological function and may produce less kids, finally a humanity will emerge which is totally immune to corona. These happen because there is tiny variations that are set into us during our birth that makes us different from our mom and dad. So producing more offspring means more variations and more chance of surviving an epidemic we have. That's why all species that had survived has a very strong urge to reproduce. The more the urge more the offspring variations and survival.

Okay let's code. So let's define a universe where only numbers that are closest to 42 exist. Why 42? Because it's the ultimate answer.

The answer to the ultimate question https://youtu.be/tK0urw144cU

So our universe is defines by this number here:

```
number = 42
```

Output:

```
42
```

This is a number we must guess using evolutionary technique.

Now our universe need to calculate the difference between the ultimate answer and a values that exist in it, so we define a error function as follows:

```
function (value, number)
    number - value
end
```

Output:

```
 (generic function with 1 method)
```

Let's test our error function:

```
□(12, 42)
```

Output:

```
30
```

Now let us experiment with random numbers. I want to generate 500 random numbers that vary from -0.5 to +0.5:

```
rand(500).- 0.5
```

Output:

```
500-element Array{Float64,1}:
  0.11811737299697067
  0.40975918563037483
  0.11512815232487483
  0.4407493702816716
 -0.06317498449812642
  0.47123272092330426
 -0.10407972969869217
  0.10466653256756975
  0.316758714503494
  0.05275897872543078
 -0.0580903153675667
 -0.0636241488226077
  0.24940396400825926
  □
  0.4043456432783936
 -0.4607426587387351
  0.4003121999197845
 -0.28198845089727187
  0.32761734763714667
 -0.35388456207140506
  0.33622477093610326
 -0.3255506303368696
 -0.04515216140695988
  0.08163723449954996
 -0.4425492244863174
  0.09909239016396998
```

Seems to work.

45.1.1. Top Survivors

Next I want to define a function called `top_survivors()` which returns the numbers that are most

closes to 42 first and least closest to 42 at the last:

```
function top_survivors(values, number, top_percent = 10)
    errors_and_values = [(abs(□(value, number)), value) for value in values]
    sorted_errors_and_values = sort(errors_and_values)
    end_number = Int(length(values) * top_percent / 100)
    sorted_errors_and_values[1:end_number]
end
```

Output:

```
top_survivors (generic function with 2 methods)
```

Lets see see in detail how `top_survivors()` is coded. First let's start with an empty function definition:

```
function top_survivors()
end
```

Now this function should receive `values` who's survival ability will be determined

```
function top_survivors(values)
end
```

The survival ability is determined by a number, the values that are closest to the number survive, so we we pass in a `number` to the function `top_survivors()`:

```
function top_survivors(values, number)
end
```

Not all values should survive, say only top 10% of the `values` that are closest to the `number` should survive, so we have a variable `top_percent` and set a default of `10` to it as shown below:

```
function top_survivors(values, number, top_percent = 10)
end
```

First we need to find errors, that is the difference between each `value` in `values`

```
function top_survivors(values, number, top_percent = 10)
    [for value in values]
end
```

compared to `number`:

```
function top_survivors(values, number, top_percent = 10)
    [(value, number) for value in values]
end
```

but there is a problem, let's say `number` is `5` and `value` is `3`, then the error is `2`, let's say another `value` is `12` hence error `(value, number` is now `-7`. Since -7 is less than 2, it does not mean that 12 is closer to 5 than 3. What we need is absolute error and we get it using the `abs()` function shown below:

```
function top_survivors(values, number, top_percent = 10)
    errors_and_values = [abs((value, number)) for value in values]
end
```

What are we going to do just with errors, let's bundle error and corresponding value into a `Tuple` using this code `(abs((value, number)), value)` as shown below:

```
function top_survivors(values, number, top_percent = 10)
    [(abs((value, number)), value) for value in values]
end
```

now let us assign it to a variable `errors_and_values`:

```
function top_survivors(values, number, top_percent = 10)
    errors_and_values = [(abs((value, number)), value) for value in values]
end
```

So we have got an Array of Tuples in `errors_and_values` where the first element in the tuple is error and second is the value that is associated with the error and it looks something like this:

```
[
  (error_1, value_1), ..... , (error_n, value_n)
]
```

Now lets sort it so that the least errors and values are at first

```
function top_survivors(values, number, top_percent = 10)
    errors_and_values = [(abs((value, number)), value) for value in values]
    sorted_errors_and_values = sort(errors_and_values)
end
```

and we store it in a variable `sorted_errors_and_values` as shown above, now all that is left is to return the `top_percent` of `errors_and_values` out, those will be our survivors, for that we calculate

the `end_number` for our array as shown in the last line of the function below:

```
function top_survivors(values, number, top_percent = 10)
    errors_and_values = [(abs(□(value, number)), value) for value in values]
    sorted_errors_and_values = sort(errors_and_values)
    end_number = Int(length(values) * top_percent / 100)
end
```

If `values` had 1943 elements then `length(value)` will be 1943. This is multiplied by `top_percent` by 100, in our case its 0.1 time 1943, that will be 194.3, and we get integer value of it result `Int(length(values) * top_percent / 100)` which is 194, this will be stored in `end_number`. The `top_percent` of `errors_and_values` will be `sorted_errors_and_values[1:end_number]` and we add this as last line to our function below which means that this will be returned out:

```
function top_survivors(values, number, top_percent = 10)
    errors_and_values = [(abs(□(value, number)), value) for value in values]
    sorted_errors_and_values = sort(errors_and_values)
    end_number = Int(length(values) * top_percent / 100)
    sorted_errors_and_values[1:end_number]
end
```

Let's test `top_survivors()` now, let me generate 500 randome numbers from -0.5 to +0.5 and see which is closest to `number`

```
survivors = top_survivors(rand(500).- 0.5, number)
```

Output:

```
50-element Array{Tuple{Float64,Float64},1}:
 (41.50295720311258, 0.49704279688742026)
 (41.503072694126054, 0.4969273058739454)
 (41.50606492163384, 0.49393507836615624)
 (41.509092656735994, 0.49090734326400853)
 (41.511967526254274, 0.4880324737457231)
 (41.51218378164256, 0.4878162183574357)
 (41.51401215336174, 0.4859878466382588)
 (41.51538950354761, 0.4846104964523883)
 (41.51687210415236, 0.4831278958476404)
 (41.520187541041686, 0.4798124589583119)
 (41.52202481756689, 0.47797518243311)
 (41.52243285351341, 0.47756714648659404)
 (41.52250385998039, 0.47749614001960716)
 ⋮
 (41.57023757877993, 0.42976242122007213)
 (41.571326533554725, 0.4286734664452725)
 (41.57377340302943, 0.4262265969705692)
 (41.57432225294512, 0.4256777470548734)
 (41.578400139207496, 0.42159986079250356)
 (41.579227438079286, 0.4207725619207161)
 (41.58325191425531, 0.41674808574469324)
 (41.58460955385501, 0.41539044614499)
 (41.5857945937749, 0.41420540622509905)
 (41.58642578719165, 0.4135742128083526)
 (41.589202493410696, 0.4107975065893039)
 (41.5916907905559, 0.40830920944409743)
```

So we see that `top_survivors()` returns top 50 closest, the first one being `(41.50295720311258, 0.49704279688742026)` that is it has an error of 41.50295720311258 and value of 0.49704279688742026.

45.1.2. Mutations

We know that 0.497 is far far from 42, so how does it inch near 42? The answer is mutation. Say that this number spawns another set of numbers that vary slightly from 0.497, some of it might be more away from 42, but some could be more near. So let's write a function to mutate a value:

```
function mutate(value, mutations = 10)
    [value + rand() - 0.5 for i in 1:mutations]
end
```

Output:

```
mutate (generic function with 2 methods)
```

Okay this `mutate()` function receives a value as first argument, and second argument is number of

children it must have which is defined my `mutations` which we have set it to a default of 10. In this statement:

```
[value + rand() - 0.5 for i in 1:mutations]
```

`value` is added with a random number anywhere between -0.5 to 0.5 and is been collected into array and its done `mutations` times. So let's test our mutate function:

```
mutate(50)
```

Output:

```
10-element Array{Float64,1}:
 49.70077118627718
 50.38719806541724
 49.659770263529325
 50.49345080427759
 49.682957552922375
 50.068482872331586
 50.418052425411894
 49.73656292165122
 50.01836131822904
 50.030683616700884
```

So see above how 50 mutates with values slightly varying from it.

I also want to introduce a function called `vcat()` that concatenates or joins two arrays as shown:

```
vcat([1, 2, 3], [4, 5, 6])
```

Output:

```
6-element Array{Int64,1}:
 1
 2
 3
 4
 5
 6
```

Our mutate function mutates only one value, but for our task we have a list of values, so let's write a function that mutates a list as shown:

```
function mutate_list(list, mutations = 10)
    output = []
    for element in list
        output = vcat(output, mutate(element, mutations))
    end
    output
end
```

Output:

```
mutate_list (generic function with 2 methods)
```

So `mutate_list` works as follows, first we have a empty function:

```
function mutate_list()
end
```

We receive a `list` which must be mutated:

```
function mutate_list(list)
end
```

Next we have a variable called `mutations` that defines the number of mutations that must take place for each element in the list:

```
function mutate_list(list, mutations = 10)
end
```

Let's have an element called output that will collect all mutations

```
function mutate_list(list, mutations = 10)
    output = []
end
```

We take each value in the `list` into a variable called `element`:

```
function mutate_list(list, mutations = 10)
    output = []
    for element in list
    end
end
```

Now we add mutated values to `output` using this statement `output = vcat(output, mutate(element, mutations))` as shown below:

```
function mutate_list(list, mutations = 10)
    output = []
    for element in list
        output = vcat(output, mutate(element, mutations))
    end
end
```

And finally once the operations is done for all elements in the list, we return the `output`:

```
function mutate_list(list, mutations = 10)
    output = []
    for element in list
        output = vcat(output, mutate(element, mutations))
    end
    output
end
```

Now let's test our `mutate()` function:

```
mutate_list((1, 2, 3, 4))
```

Output:

```
40-element Array{Any,1}:
 1.1981180737202284
 0.6826820656478603
 1.4263346623043727
 1.0445431960263634
 0.9986709617577949
 0.5041082408398894
 1.206211947133361
 0.5703313900822442
 0.7085107623470692
 0.6808788103524246
 1.8926516209570248
 2.3122314617476407
 2.4123045970406345
 
 2.5112031422146033
 3.0098577316693103
 3.5139755069699214
 3.5552827017499755
 3.632721017569514
 3.6865403468441524
 3.908433918456275
 4.330548003821807
 3.8199880565807103
 4.4004408963825155
 3.8622190204543214
 4.299693796909806
```

So we have condition of best survival defined by `number = 42`, then we have a function called `top_survivors()` which will clean out values it thinks are unfit and have only those that meet the top percent of the criteria, then we have a function `mutate()` that will change the numbers. So we have made the bits and pieces of our universe namely:

1. Condition for best fit `number = 42`
2. Way to clean those that are not fitting well`top_survivors()`
3. A way to change `mutate()`

45.1.3. Creating our universe

Now we have to put this all together. So first let us have a initial set of values, a bunch of numbers to start with:

```
# let there be initial values
initial_values = rand(500)
```

Figure 9. Brahma the initiator

First all those values are survivors, so let us define `survivors` and assign it to `initial_values`:

```
# let there be initial values
initial_values = rand(500)
survivors = initial_values
```

Let's define a variable called `generations` that will hold an value of the maximum number of generations allowed in our universe:

```
# let there be initial values
initial_values = rand(500)
survivors = initial_values
generations = 500
```

Next for plotting purpose, for us to see what happened visually we sample and store values / numbers that are created in each generation that matches closest to `number`, that is 42 in our case, we store that sampled values in `top_survivors_sample`

```
# let there be initial values
initial_values = rand(500)
survivors = initial_values
generations = 500
top_survivors_sample = []
```

Now for each generation:

```
# let there be initial values
initial_values = rand(500)
survivors = initial_values
generations = 500
top_survivors_sample = []

for generation in 1:generations
end
```

we make the survivors mutate and create offspring:

```
# let there be initial values
initial_values = rand(500)
survivors = initial_values
generations = 500
top_survivors_sample = []

for generation in 1:generations
    survivors = mutate_list(survivors)
end
```

Figure 10. Shiva the god of Destruction

now the force of destruction comes to play only those that are closet to ideal condition are selected in this line `errors_and_values = top_survivors(survivors, number)` others are eliminated:

```
# let there be initial values
initial_values = rand(500)
survivors = initial_values
generations = 500
top_survivors_sample = []

for generation in 1:generations
    survivors = mutate_list(survivors)
    errors_and_values = top_survivors(survivors, number)
end
```

Figure 11. Vishnu the God that saves and ensures continuity

now we salvage only the values for the next generation in the following statement `survivors = [value for (error, value) in errors_and_values]` as shown below:

```
# let there be initial values
initial_values = rand(500)
survivors = initial_values
generations = 500
top_survivors_sample = []

for generation in 1:generations
    survivors = mutate_list(survivors)
    errors_and_values = top_survivors(survivors, number)
    survivors = [value for (error, value) in errors_and_values]
end
```

Figure 12. Saraswathi the God of knowledge and learning

Now we record the top 10 survivors into our sample array `top_survivors_sample` in this statement `push!(top_survivors_sample, survivors[1:10])`, this will help us to plot and understand what happened:

```
# let there be initial values
initial_values = rand(500)
survivors = initial_values
generations = 500
top_survivors_sample = []

for generation in 1:generations
    survivors = mutate_list(survivors)
    errors_and_values = top_survivors(survivors, number)
    survivors = [value for (error, value) in errors_and_values]
    push!(top_survivors_sample, survivors[1:10])
end
```

And for the first and every 10th generation we print out the top values using these lines

```
if (generation == 1) || (generation % 10 == 0)
    println(generation, " => ", survivors[1:5])
end
```

So you can see the completed code below:

```
# let there be initial values
initial_values = rand(500)
survivors = initial_values
generations = 500
top_survivors_sample = []

for generation in 1:generations
    survivors = mutate_list(survivors)
    errors_and_values = top_survivors(survivors, number)
    survivors = [value for (error, value) in errors_and_values]
    push!(top_survivors_sample, survivors[1:10])

    if (generation == 1) || (generation % 10 == 0)
        println(generation, " => ", survivors[1:5])
    end
end
```

Output:

```
1 => [1.489963570249114, 1.4748790114913484, 1.4622739648861949, 1.4510704237615362,
1.4490538800141282]
10 => [5.660855018791918, 5.598888609438815, 5.593011278753805, 5.5762836556003545,
5.566632716837373]
20 => [10.24997392360673, 10.215807815922725, 10.21448723688237, 10.214464268619894,
10.204137257813436]
30 => [14.909055039342904, 14.889731189243328, 14.884161682306756, 14.874322175303265,
```

```
14.867230337010271]
40 => [19.52250107015328, 19.50132983259004, 19.48141475284547, 19.47896089764954,
19.478514865080626]
50 => [24.185076343905422, 24.171229043560263, 24.16911898120017, 24.168797157127305,
24.151685651797276]
60 => [28.777004719428835, 28.73967040390202, 28.730165929867656, 28.721490919382102,
28.705793402309293]
70 => [33.38810260130008, 33.3806820730437, 33.35186837385183, 33.34216038995459,
33.32739352190749]
80 => [37.931008629223165, 37.9235457841195, 37.91415226790492, 37.91195802545634,
37.91106132860069]
90 => [41.99992407747299, 42.00008624788804, 41.99991137236084, 41.99979061755105,
42.000233034952544]
100 => [42.000123781083616, 42.000235748738795, 42.000307239484606, 41.99960275074321,
41.99956774943434]
110 => [42.000020774696715, 41.99997835399716, 41.99993951603409, 41.99979201596632,
41.999598390078326]
120 => [42.000244420648414, 41.999688461201856, 42.00033212168877, 42.000419165315776,
41.999043587401665]
130 => [42.00000316583509, 41.99980027149457, 42.00043079801067, 41.99954988980232,
41.99950264181341]
140 => [41.999923715753546, 41.99977377760766, 42.00033523277993, 41.99951474643227,
42.00055740776272]
150 => [41.99999977643734, 42.00037448792328, 42.00045229876902, 41.99954594667108,
42.000492403893965]
160 => [41.999937769558926, 41.999854484705295, 41.999819702244785, 42.00025056814451,
42.00039399008882]
170 => [42.00005547930738, 41.99979529808799, 41.99972233935926, 41.99971873415906,
41.99963575444441]
180 => [42.00003109415735, 42.000200426218896, 41.999755219184195, 42.00043618008025,
41.999501915890455]
190 => [41.99985969909203, 42.00021165635373, 41.99976762837727, 42.00031788049009,
42.000332341607766]
200 => [41.99992962001092, 42.00017098936294, 41.99979094577462, 42.00030728974183,
41.999650626630036]
210 => [42.00020298104202, 41.99975261684113, 41.99975023469254, 41.99957908132207,
41.999445950654746]
220 => [41.99992584076578, 41.999855258167386, 42.000200104286066, 41.999702460834605,
42.00035077159908]
230 => [42.00035209859171, 41.99963549012812, 42.00037216436359, 41.99951880529105,
42.00050803530517]
240 => [41.99984779759352, 41.99974134448669, 42.00031532262816, 41.99962683164088,
42.00052285058658]
250 => [41.999728429745, 42.000393543238104, 42.00051227111797, 41.999461252932434,
42.00064395532183]
260 => [42.00025814042138, 42.00031589894168, 42.00041985364005, 41.99956313819255,
41.9993910261099]
270 => [41.99999510362798, 42.00001999291032, 41.9999579395399, 41.999948699620724,
42.00035888305544]
280 => [42.0000349216614, 41.99994198625842, 41.99991656164299, 42.00016283910495,
41.99974913725434]
```

```
290 => [42.000088552147226, 42.00009177413705, 42.000285959629764, 41.999368276803075,
42.00085092034136]
300 => [42.00002153736082, 42.00017318821541, 42.00027218263787, 42.000360859044456,
41.99956805018513]
310 => [41.99982214130841, 41.99967598916952, 42.00033996524177, 41.99959063668834,
42.000454542266226]
320 => [42.00006126086911, 42.00008176129088, 41.99991628417523, 41.99984827263701,
42.0006948512896]
330 => [41.999920931176746, 42.000084474769224, 42.00011049825621, 42.0001780995309,
41.999746240047266]
340 => [42.00016462077862, 41.99973103912161, 41.99972713436191, 42.000314926474644,
41.99954583326124]
350 => [41.999909064720605, 41.99989664326481, 42.00012807380437, 41.99981089152715,
42.00044965487015]
360 => [41.99998035859879, 41.999741519792536, 41.99972834361312, 42.000466246698764,
41.999445378979175]
370 => [41.99998473751336, 42.000118961242094, 42.00019927235091, 41.99967103681347,
42.0003893406721]
380 => [42.00004994314298, 42.000088485492014, 41.99989507939007, 42.00021089501884,
41.999768499453445]
390 => [42.00016741179328, 41.99983192017003, 42.00019597710459, 42.000246488574184,
41.9997500616498]
400 => [42.0000690034192, 42.00032210252985, 42.00044477418595, 41.99950447485576,
41.999409987775174]
410 => [41.99984997793723, 42.00021601004096, 41.999733823597175, 41.99943005559297,
42.000572405921815]
420 => [41.99978319192324, 42.0003312867216, 41.99956076412637, 42.000495777523554,
42.00057076477468]
430 => [41.9999969877506, 42.00015622715756, 41.99978672785641, 42.00025219550749,
41.999745559762715]
440 => [41.999961313124366, 41.99992889660888, 42.00022494455986, 42.00027211260345,
41.99971622361044]
450 => [41.99992660449093, 42.00012955558257, 41.99986339510042, 42.00018388328064,
41.99973756767785]
460 => [41.99980983221668, 41.99979169174335, 41.99960684933666, 41.99945475746643,
42.00056035870122]
470 => [42.000037583752786, 42.000103148938535, 42.000148858716024,
41.999803261589705, 41.99958494029524]
480 => [42.00010180604779, 41.999839312694654, 42.000441302743766, 42.000564805088516,
42.00069368757735]
490 => [41.999991513319785, 42.000069257104975, 41.99971242785191, 42.000289368310895,
41.99933899177168]
500 => [42.00014966355293, 41.99981721794562, 42.00030285516861, 42.000342118257144,
41.99949493404719]
```

As you can see above as the index progresses, the top 5 values approach 42, sometimes they shoot slightly above 42, but they are limited by the universal `number` that determines the survival ability, so those which shoot up too much are also eliminate.

45.1.4. Plotting what we had done

Let's plot and learn from stored values in `top_survivors_sample`. First let's have a range for plotting that spans from 1 to `generations`:

```
plotting_range = 1 : generations
```

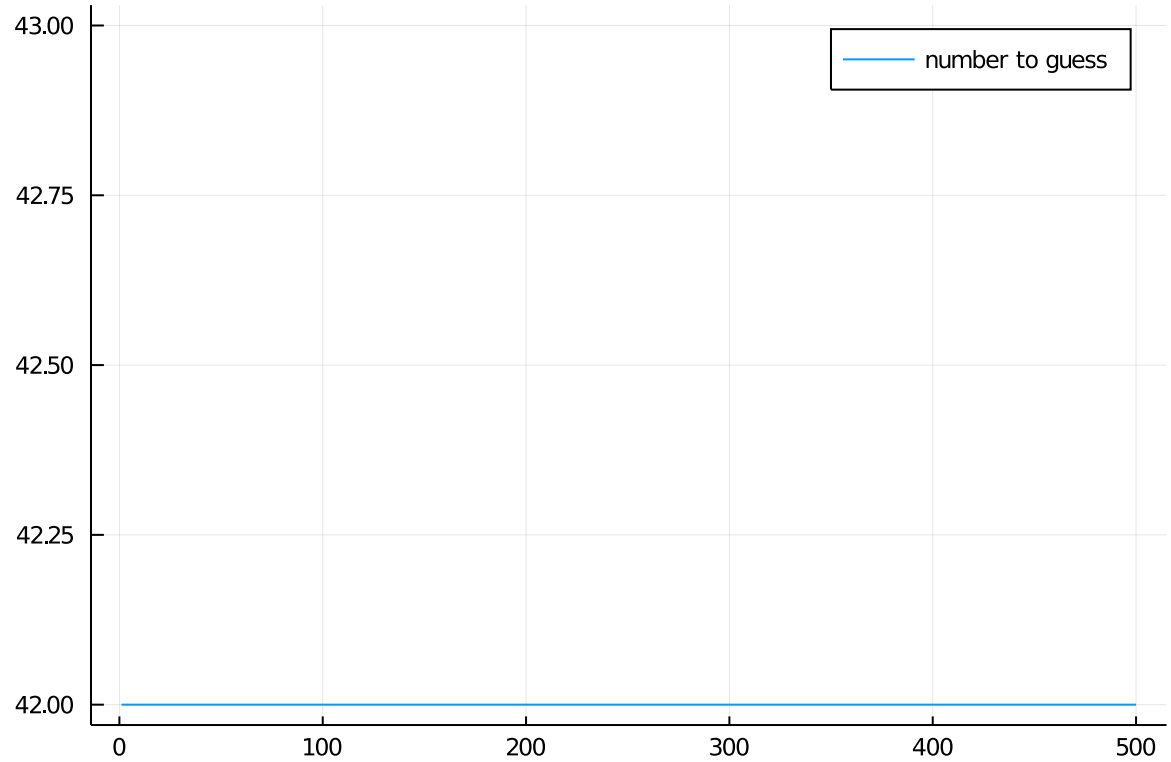
Output:

```
1:500
```

Next let's plot the target, which is nothing but `number`, we create a array having values that equal `number` and is `generations` long using this code `fill(number, generations)`, `fill()` creates an array of `generations` long and fills it with `number`, we pass it as second argument to `plot()` as shown:

```
using Plots
progress_plot = plot(plotting_range, fill(number, generations), label = "number to
guess")
```

Output:



`plot()` by default does a line plot, so we get a straight line at 42 as shown above. We store the plot in a variable named `progress_plot`.

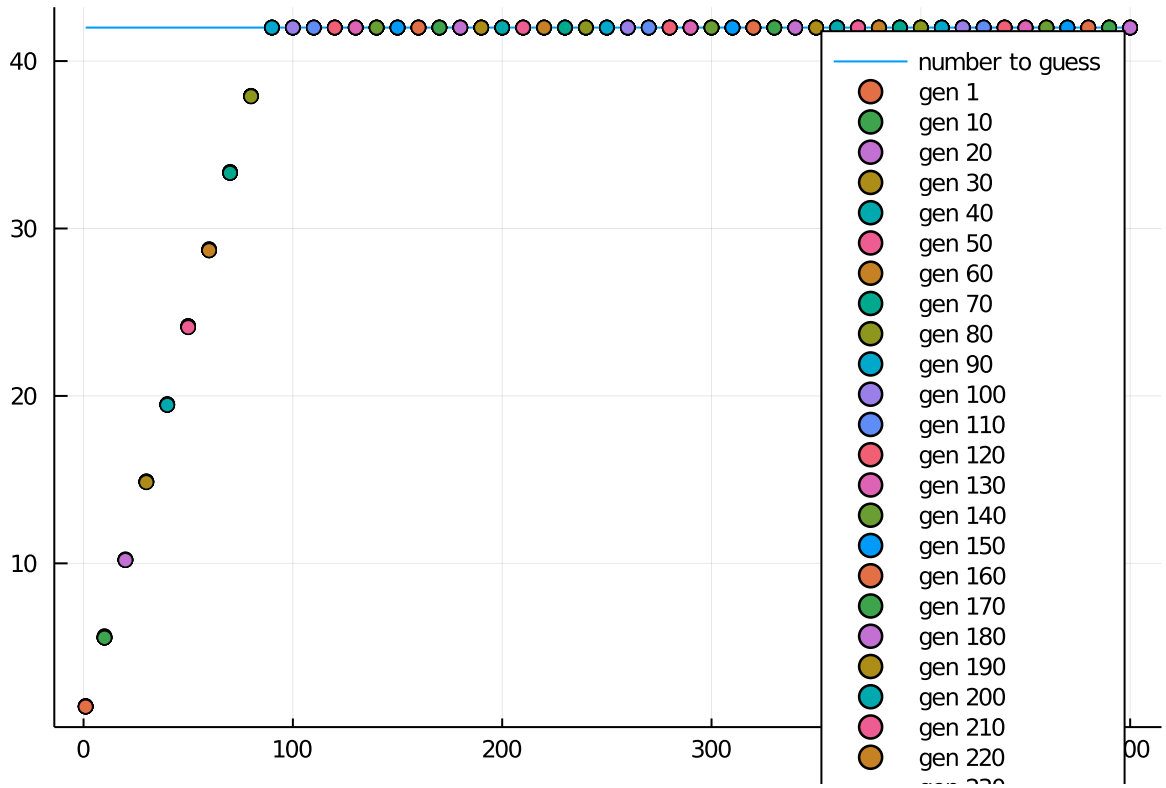
In the code below, for the 1st and every 10th generation, we append `progress_plot` with ten values of `i` that is the generation value using this code `fill(i, 10)` for x axis and on the y axis we plot the

corresponding value of top survivors for that generation `top_survivors_sample[i]` as shown below:

```
for i in plotting_range
    if (i == 1) || (i % 10 == 0)
        plot!(progress_plot, fill(i, 10), top_survivors_sample[i], label = "gen $i",
seriestype = :scatter)
    end
end

progress_plot
```

Output:



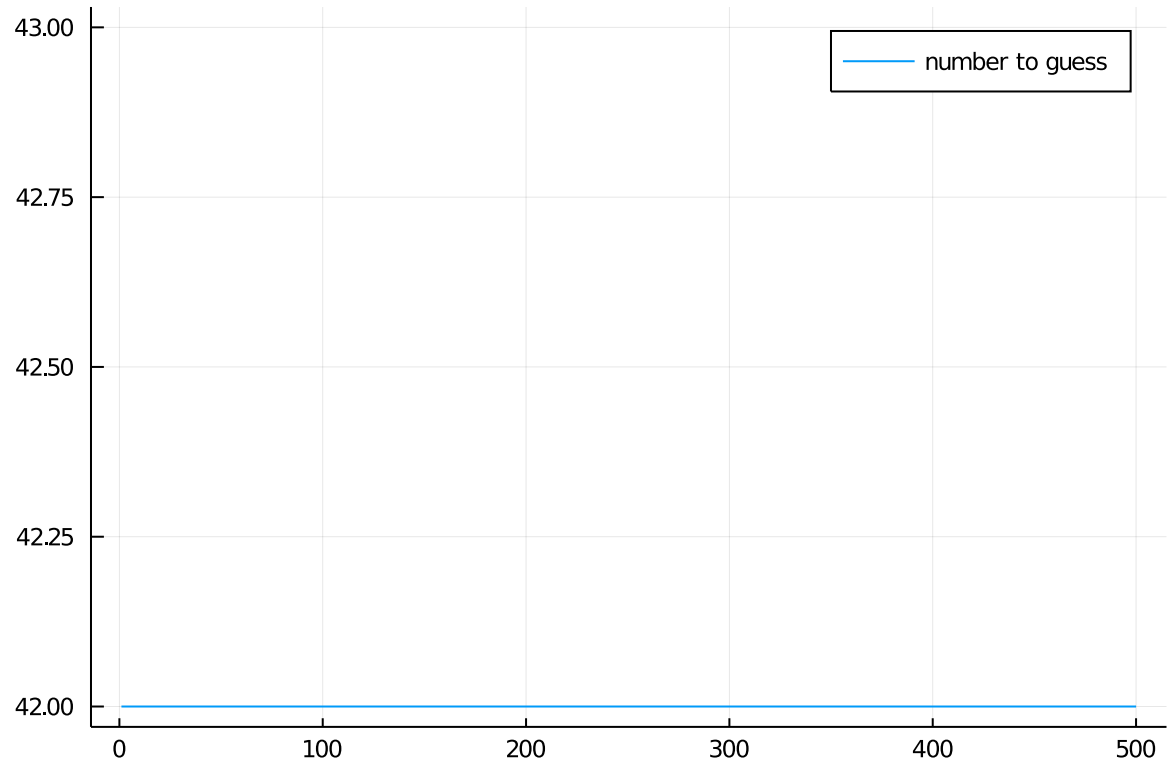
So as we see above, some where around the 100th generation we are reaching the targeted 42, why don't you raise the value of `number` in the notebook 71 and see when are we reaching the target, can you say why? Can you increase the number to -7658 and see if we are able to reach it? If yes why and if not why not?

Chapter 46. Fine grained plot

`top_survivors_sample` is an array, but since the plot dimensions above are so huge we are unable to see it, so let's do a fine grained plot, so we plot the `number` first:

```
fine_grain_plot = plot(plotting_range, fill(number, generations), label = "number to guess")
```

Output:

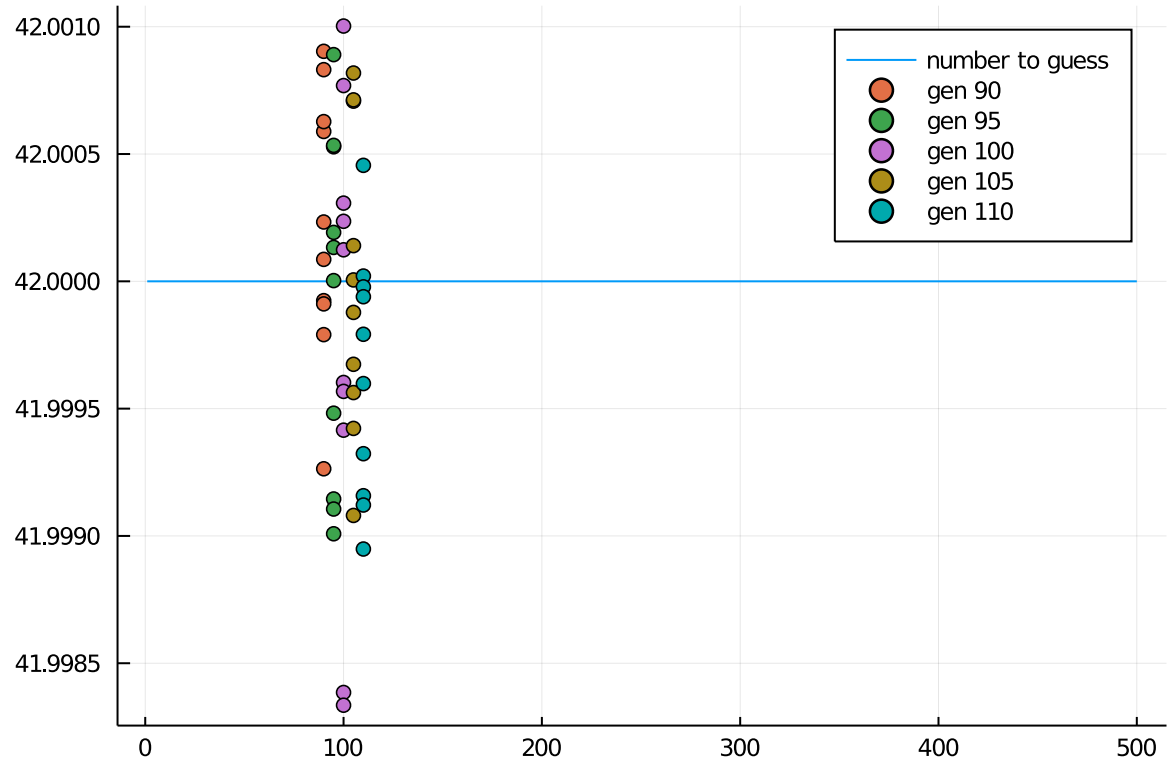


Next just from generation 90 to 110 we plot the `top_survivors_sample` as shown below:

```
for i in 90:110
    if (i % 5 == 0)
        plot!(fine_grain_plot, fill(i, 10), top_survivors_sample[i], label = "gen $i", seriestype = :scatter)
    end
end

fine_grain_plot
```

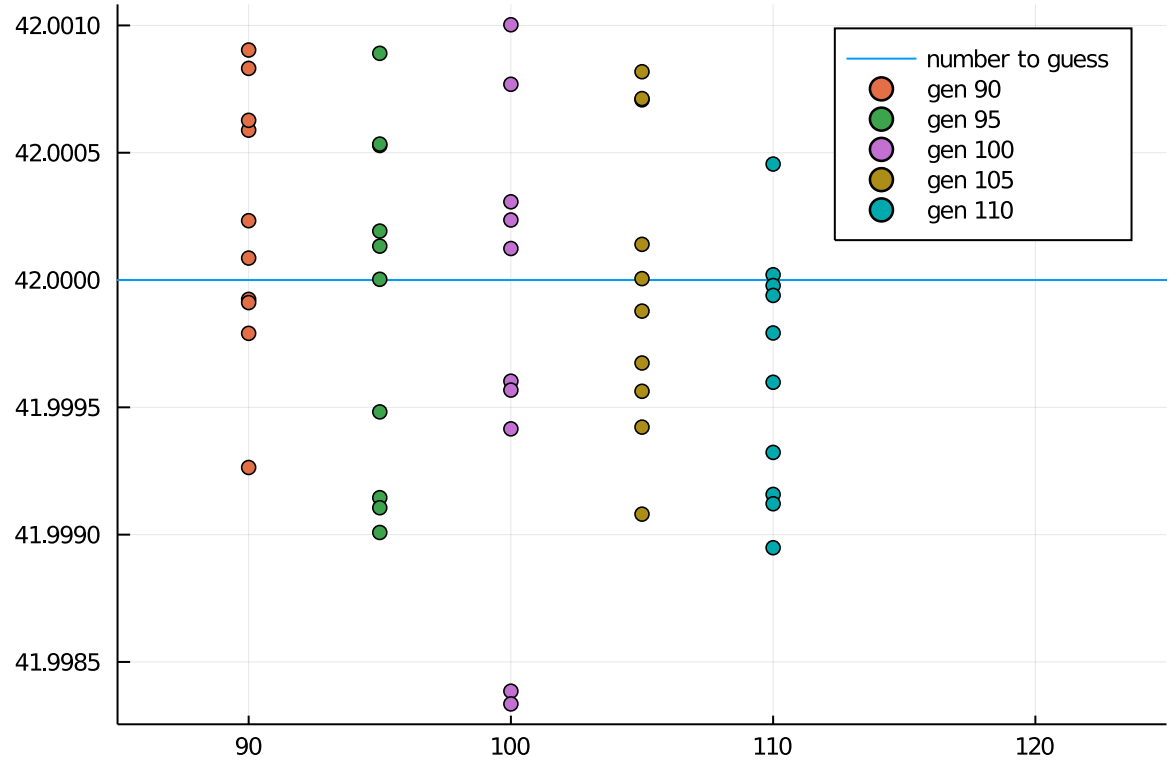
Output:



Let's enlarge the plot so that we can see better, for that let's zoom in, we set x-limits `xlims` from 85 to 125:

```
plot!(fine_grain_plot, xlims = (85, 125))
```

Output:



So you can see how the generations vary in value, and how they hover near 42.

The Jupyter notebook for this section is here https://gitlab.com/datascience-book/code/-/blob/master/genetic_algorithm_to_guess_number.ipynb.

Figure 13. Lakshmi - Goddess of comfort, wealth and progress you gain by your experience

46.1. Curve fitting with genetic algorithm

Video lecture for this section could be found here https://youtu.be/OLSzsGRLCKY

Before reading this section it would be great if you could read and get familiarized with Guessing a Number with Genetic Algorithm.

In this section we would like to fit a line with genetic algorithm for the equation $y = 7x + 5$. So let's write the equation in Julia

```
y(x) = 7x + 5
```

Output:

```
y (generic function with 1 method)
```

Now let's assign x to range 1 to 5 in steps of 0.1

```
x = 0 : 0.1 : 5
```

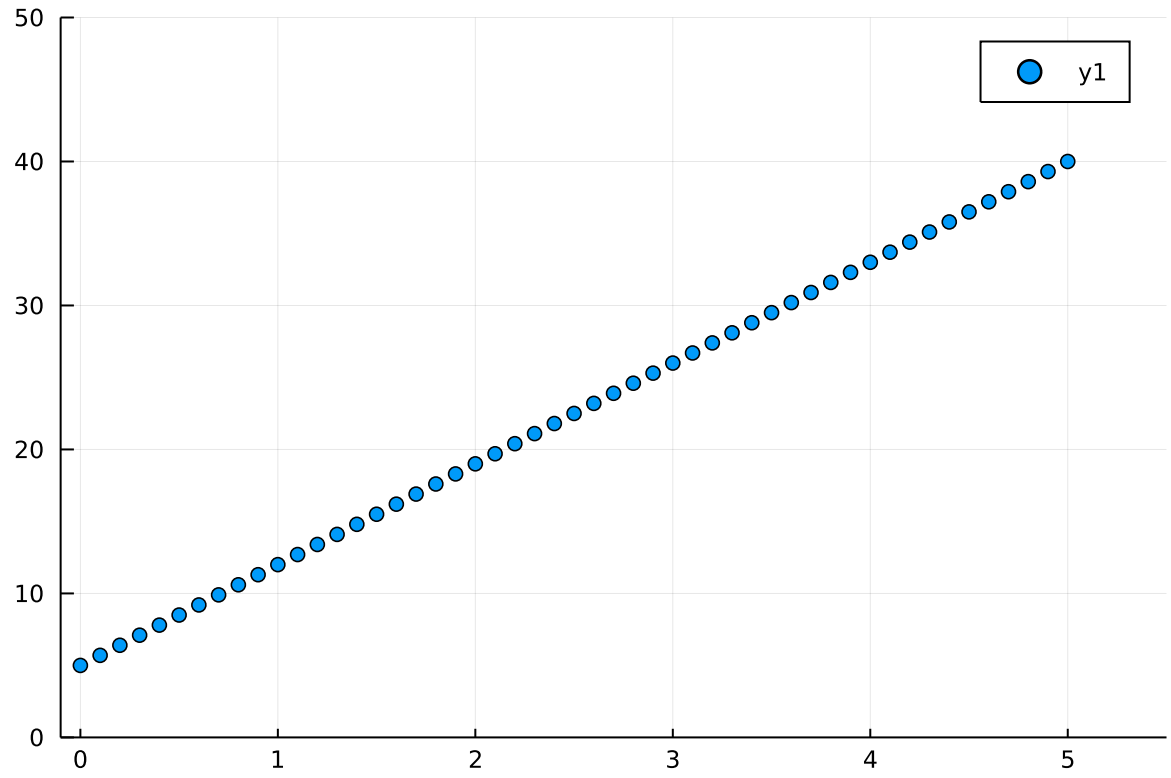
Output:

```
0.0:0.1:5.0
```

Now let's plot the line

```
using Plots
```

```
scatter_plot = scatter(x, y.(x), xlims = (-0.1, 5.5), ylims = (0, 50))
```



There is a property in Julia which I want you to remember, let's say we have a tuple `(5, 7)` and we assign it to two variables `m` and `c` as shown below

```
m, c = (5, 7)
println("m = ", m)
println("c = ", c)
```

Output:

```
m = 5
c = 7
```

Then the first variable will take the first value of the tuple and second will take the second value. We will be using this technique here.

For genetic algorithm, mutation or change is very important. Let's write a mutation function for a two valued tuple as shown:

```
function mutate(value, mutations = 10)
    [value + rand() - 0.5 for i in 1:mutations]
end

function mutate(mc::Tuple, number_of_mutations = 10)
    m, c = mc
    ms = mutate(m, number_of_mutations)
    cs = mutate(c, number_of_mutations)

    [(ms[i], cs[i]) for i in 1:number_of_mutations]
end
```

Output:

```
mutate (generic function with 4 methods)
```

and we test that function:

```
mutate((1, 2))
```

```
10-element Vector{Tuple{Float64, Float64}}:
 (1.1205168123693385, 2.41514037266611)
 (0.5079882676843603, 1.8217999723149112)
 (1.1581739571074665, 2.459460436379917)
 (0.5796738025677814, 1.9702636743774655)
 (1.0321606616260794, 1.7632954510809746)
 (1.3545156402167784, 1.8782505949943635)
 (1.3407759822406362, 2.3597713445300306)
 (0.8524954649600578, 2.4966166282430047)
 (0.991673117157789, 2.4749204653332812)
 (0.9933905544923662, 1.5623361787716572)
```

Now let me explain how it works. First we have a mutation function that takes single value and mutates it as shown below

```
function mutate(value, mutations = 10)
    [value + rand() - 0.5 for i in 1:mutations]
end
```

Now we kind of want to extend this `mutate` function to tuple, so we write a declaration for it as shown

```
function mutate(value, mutations = 10)
    [value + rand() - 0.5 for i in 1:mutations]
end

function mutate(mc::Tuple, number_of_mutations = 10)
end
```

Note that unlike the `mutate(value, mutations = 10)` this `mutate(mc::Tuple, number_of_mutations = 10)` clearly says it needs a tuple. Now imagine I pass a tuple to `mutate` like `mutate1, 2`, clearly Julia will pass it to `mutate(mc::Tuple, number_of_mutations = 10)` and not to `mutate(value, mutations = 10)`. So now let's define the tuple `mutate`.

Let's assume the tuple will have only two values, so the first value should be assigned to a variable `m` and second one to `c` as shown

```
function mutate(mc::Tuple, number_of_mutations = 10)
    m, c = mc
end
```

Now all we need to do is to mutate `m` and `c` by `number_of_mutations` times and combine it. To mutate `m` and `c` we use the following code

```
ms = mutate(m, number_of_mutations)
cs = mutate(c, number_of_mutations)
```

In the above code `mutate(m, number_of_mutations)` and `mutate(c, number_of_mutations)` calls `mutate(value, mutations = 10)`. We store the values of mutation in variable `ms`, I pronounce it as emmmss and other in seas `cs`, let's plugin this code into our function as shown

```
function mutate(mc::Tuple, number_of_mutations = 10)
    m, c = mc
    ms = mutate(m, number_of_mutations)
    cs = mutate(c, number_of_mutations)
end
```

Now we combine these mutations to form a `Array` of `Tuple's and return it as shown

```
function mutate(mc::Tuple, number_of_mutations = 10)
    m, c = mc
    ms = mutate(m, number_of_mutations)
    cs = mutate(c, number_of_mutations)

    [(ms[i], cs[i]) for i in 1:number_of_mutations]
end
```

Now let's write an error function, let's say that we have `m` and `c` that our algorithm recommends, for a given `x`, the prediction will be `(m * x + c) `, and in the training phase the value of `y` may vary from our prediction, so, we need to minus `y` from our prediction to get error as shown:

```
(m, c, x, y) = (m * x + c) - y
```

Output:

```
 (generic function with 1 method)
```

Let's say our `m` is 5, and `c` is 4, let's see what's out error when `x` is 10 as shown below

```
(5, 4, 10, y(10))
```

Output:

```
-21
```

Now some one says he has 10 values of x and corresponding y with him, he wants us to fit a line to

predict the unknown stuff, we decide a value of m and c, we need to know the total error for all values of our prediction, so let's write a `total_error` function as shown:

```
function total_error(m, c, x, y)
    ΣΔ = 0

    for i in 1:length(x)
        ΣΔ += abs(□(m, c, x[i], y[i]))
    end

    ΣΔ
end
```

output:

```
total_error (generic function with 1 method)
```

In the `total_error`, we have a variable `ΣΔ` to store the total error, we assign it to zero as shown:

```
function total_error(m, c, x, y)
    ΣΔ = 0
end
```

now for each element if `x`

```
function total_error(m, c, x, y)
    ΣΔ = 0

    for i in 1:length(x)
    end
end
```

we add absolute value of error to total error variable `ΣΔ`

```
function total_error(m, c, x, y)
    ΣΔ = 0

    for i in 1:length(x)
        ΣΔ += abs(□(m, c, x[i], y[i]))
    end
end
```

we return the total error variable `ΣΔ`

```
function total_error(m, c, x, y)
    ΣΔ = 0

    for i in 1:length(x)
        ΣΔ += abs(□(m, c, x[i], y[i]))
    end

    ΣΔ
end
```

Now let's check the `total_error` function

```
total_error(5, 4, [1, 2, 3, 4, 5], [1, 2, 3, 4, 5])
```

Output:

```
80
```

Seems to work!

Let's now write our `top_survivors` function, this one takes `Array` of m's and c's `Tuple□s as `mcs` (I spell it as *emm seee's*), an array of training inputs namely `x_train` and `y_train`, and selects the `top_percent` of `mcs` that that deviates the least with `y_train` for the given `x_train`.

```
function top_survivors(mcs, x_train, y_train, top_percent = 10)
    errors_and_values = []

    for mc in mcs
        m, c = mc
        error = total_error(m, c, x_train, y_train)
        push!(errors_and_values, (error, mc))
    end


    sorted_errors_and_values = sort(errors_and_values)
    end_number = Int(length(mcs) * top_percent / 100)
    sorted_errors_and_values[1:end_number]
end
```

Output:

```
top_survivors (generic function with 2 methods)
```

I would encourage you to read my previous Guessing a Number Genetic Algorithm section to understand `top_survivors` function.

Now let's setup the needed inputs for `top_survivors`, let's have initial `mcs` as `(0, 0)`:

```
x_train = [1, 2, 3, 4, 5]
y_train = y.(x_train)
mcs = mutate((0, 0))
```

Output:

```
10-element Vector{Tuple{Float64, Float64}}:
 (0.16087862232389538, 0.018092305432591882)
 (0.08312155751992667, -0.022630590705241538)
 (-0.32885966826028445, -0.4936484604469795)
 (-0.19041016854936288, 0.4707757045419352)
 (0.011998626752193209, -0.276041600093212)
 (-0.18351337891221542, -0.31274629114220875)
 (0.014823442262999142, 0.03761731346306618)
 (-0.19617757028312166, -0.11227454061420317)
 (0.07142332361410042, -0.2765731664220268)
 (0.3720719485797561, 0.44332686459739223)
```

Let's test it out:

```
top_survivors(mcs, x_train, y_train)
```

Output:

```
1-element Vector{Any}:
 (122.2022864483167, (0.3720719485797561, 0.44332686459739223))
```

Looks like our `m` of 0.3720719485797561 and `c` of 0.44332686459739223 are the fittest with an error of `122.2022864483167`. Now all we need to do it to tell it to select top survivors across many generations so that we get a near perfect fit as shown:

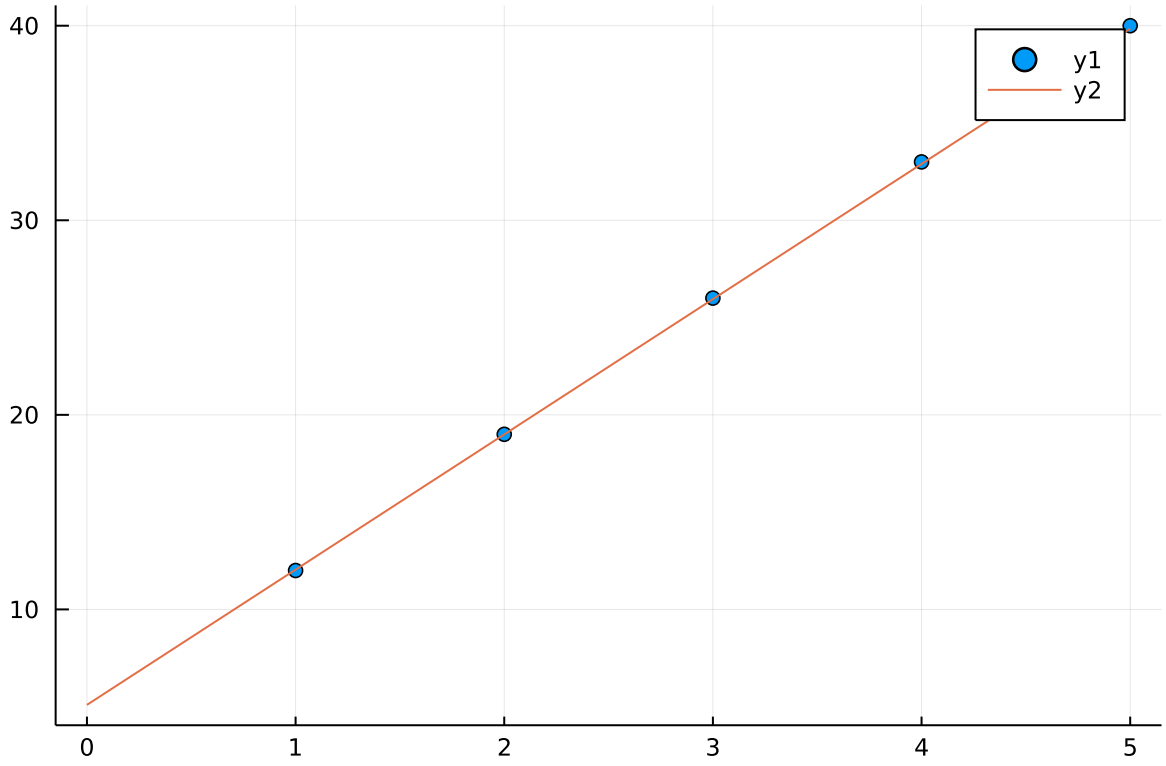
```
generations = 40

top_survivor = Nothing
mc = (0, 0)

for i in 1:generations
    mcs = mutate(mc)
    top_survivor = top_survivors(mcs, x_train, y_train)[1]
    _error, mc = top_survivor
end

hm, hc = mc
p = scatter(x_train, y_train)
h(x) = hm * x + hc
plot!(p, x, h.(x))
```

Output:



The code below is almost the same as code above, but note the `@gif` which collects the plot images into a gif file so that we can see it as animation:

```
generations = 40

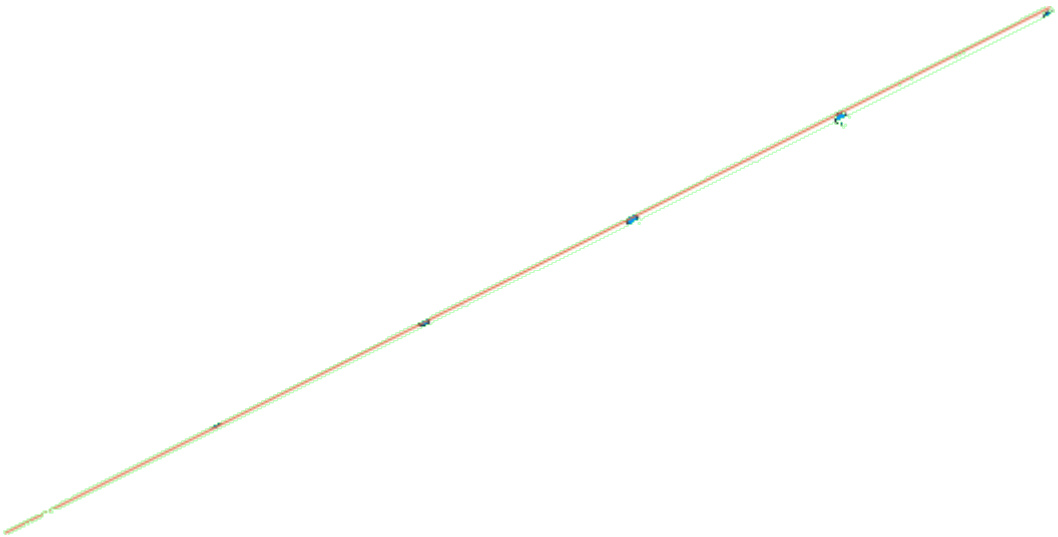
top_survivor = Nothing
mc = (0, 0)

@gif for i in 1:generations
    mcs = mutate(mc)
    top_survivor = top_survivors(mcs, x_train, y_train)[1]
    _error, mc = top_survivor

    hm, hc = mc
    p = scatter(x_train, y_train)
    h(x) = hm * x + hc
    plot!(p, x, h.(x), ylims = (0, 50))
end
```

Output:

```
┌ Info: Saved animation to
│   fn = /Users/mindaslab/data_science_with_julia/code/tmp.gif
└ @ Plots /Users/mindaslab/.julia/packages/Plots/g581z/src/animation.jl:104
```



See how in 40 generations our algorithms nearly fits a line to `y(x) = 7x + 5`, you may change 7 and 5 in `y(x)` to any thing and our algorithm in successive generations will try to fit line to a bunch of points better and better in successive generations.

When I tried, this was the final value I got for m and c

```
mc
```

Output:

```
(6.838106002778459, 5.334256635688782)
```

In your computer it could vary, but the line will be a near perfect fit for the bunch of points.

You can get the code for this section here https://gitlab.com/datascience-book/code/-/blob/master/genetic_alogorithms_line_fitting.ipynb.

Chapter 47. K Nearest Neighbors

Chapter 48. Decision Tree

Decision tree is primary used for classification. In this section let's see how to use decision tree to classify the Titanic data set https://gitlab.com/datascience-book/code/-/blob/master/data/titanic.csv.

48.1. Understanding the Titanic data set

Let's first understand the Titanic data set, if you take a look at it here https://gitlab.com/datascience-book/code/-/blob/master/data/titanic.csv , you would not that it's a CSV file which has the following columns:

Column Name	Explanation
PassengerId	A unique number given to every row of data.
Survived	If it's 1, then it means the passenger survived the wreak, if 0 then it means he or she died.
Pclass	the passenger class, whether the passenger is occupied in 1st, second or third class.
Name	Self explanatory
Sex	Self explanatory
Age	Self explanatory
SibSp	Number of sibling or spouse of the passenger who is on board.
Parch	Number of parents or children of the passenger onboard the ship.
Ticket	The ticket number.
Fare	The fare paid by the passenger, possibly in British pounds.
Cabin	Name or number of the cabin where the passenger stays.
Embarked	From which place the passenger embarked the the ship.

With the the knowledge from the above table let's proceed.

48.2. Entropy

Get the Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/entropy.ipynb

Video lecture for this section could be found here https://youtu.be/d3m9m8KhU9Y

First let's look into what's entropy. In short entropy is a measure of impurity in a system. Let me explain. Let's fire our Jupyter notebook and import our `ml_lib.jl`

```
include("ml_lib.jl")
```

Now the measure of impurity, that is entropy is given by the following formula:

$$H(x) = -\sum p(x) \cdot \log_2(p(x)$$

That is entropy is denoted by $H(x)$ and the probability of an element x found in a set is given by $p(x)$, then the entropy is given by the above formula. Note that we use log two above.

I am not sure we can use natural log or others, let me clarify it when I feel so.

So log to the base two in Julia is computed by a function `log2`, let's find log two of 8:

```
log2(8)
```

Output:

```
3.0
```

Let's dive in and calculate the entropy of the vector shown below:

```
fruits = ["apple", "apple", "apple", "apple"]
```

So the `fruits` just contains 4 elements of `"apple"` , hence it has got no impurity in it. Which means its entropy should be really low. As though we don't belive what we have type let's count the number of occurrences of `"apple"` in fruits:

```
counter(fruits)
```

Output:

```
Dict{Any, Any} with 1 entry:
  "apple" => 4
```

So there are 4 elements in `fruits` and all of them are `"apple"`, so let's calculate the probability of `"apple"` which is:

```
p_apple = 4 / 4
```

Output:

```
1.0
```

Now let's calculate the entropy of `"apple"`:

```
entropy_1 = - 1 * log2(1)
```

Output:

```
-0.0
```

Since this set contains just pure `"apple"` and nothing else, it's entropy is the least, that is zero.

Now let's add an element `"orange"` to the set:

```
fruits = ["apple", "apple", "apple", "apple", "orange"]
```

Output:

```
5-element Vector{String}:
 "apple"
 "apple"
 "apple"
 "apple"
 "orange"
```

Now the probability of `"apple"` becomes:

```
p_apple = 4 / 5
```

Output:

```
0.8
```

And the probability of `"orange"` becomes:

```
p_orange = 1 / 5
```

Output:

```
0.2
```

Now let's calculate it's entropy:

```
entropy_2 = -(0.8 * log2(0.8)) - (0.2 * log2(0.2))
```

Output:

```
0.7219280948873623
```

Definitely it's higher this time, since `"orange"` is an impurity in a set of `"apple"`.

Now assuming there are equal number of elements ina a set, so the proablity is half, let's calculate the entropy and see:

```
entropy_3 = -(0.5 * log2(0.5)) - (0.5 * log2(0.5))
```

Output:

```
1.0
```

The entropy is even higher than 0.72 as the set is more mixed up.

Now let's say we have a set of 10 fruits, let's say we have 3 oranges, 3 sappota's and 4 apples, now let's calculate the entropy:

```
entropy_4 = -(0.3 * log2(0.3)) - (0.3 * log2(0.3)) - (0.4 * log2(0.4))
```

Output:

```
1.5709505944546684
```

In `ml_lib.jl`, I have coded a function called `element_probabilities`, ehich when given a vector, puts out he probabilities of elements, let's try that out on our `fruits` vector:

```
element_probabilities(fruits)
```

Output:

```
Dict{Any, Any} with 2 entries:
  "orange" => 0.2
  "apple"  => 0.8
```

I have also coded a function called `entropy` in `ml_lib.jl` which when passed a vector calculates it's entropy, let's use it to find the entropy of `fruits`

```
entropy(fruits)
```

Output:

```
0.7219280948873623
```

Seems to work!

The reader is encouraged to refer source of `ml_lib.jl` which could be found here https://gitlab.com/datascience-book/code/-/blob/master/lib/ml_lib.jl

48.3. Applying Entropy on Titanic Dataset

Get the Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/entropy_on_titanic.ipynb and here https://gitlab.com/datascience-book/code/-/blob/master/entropy_on_titanic_feature_engineered.ipynb

In[1]:

```
include("ml_lib.jl")
```

Out[1]:

```
entropy (generic function with 1 method)
```

In[2]:

```
using DataFrames
using CSV
```

In[3]:

```
df = DataFrame(CSV.File("data/titanic.csv"))

# Get the non missing rows
df_non_missing = dropmissing(df)

# Get only adults
filter_adults = df_non_missing.Age .>= 18
df_adults = df_non_missing[filter_adults, :]

# Get only the men
filter_men = df_adults.Sex .== "male"
df_men = df_adults[filter_men, [:Sex, :Age, :Survived, :Pclass]]

# Get only the woman
filter_women = .!filter_men
df_women = df_adults[filter_women, [:Sex, :Age, :Survived, :Pclass]]

# Get only children
filter_children = .!filter_adults
df_children = df_non_missing[filter_children, [:Age, :Survived]]
```

Output:

	Age	Survived
1	4.0	1
2	1.0	1
3	3.0	1
4	2.0	0
5	2.0	0
6	0.92	1
7	17.0	1
8	16.0	1
9	2.0	1
10	14.0	1
11	4.0	1
12	16.0	1
13	17.0	1
14	4.0	1
15	15.0	1
16	6.0	1
17	17.0	1

	Age	Survived
18	11.0	1
19	16.0	1

In[4]:

```
entropy([1, 1, 1, 1, 1])
```

Out[4]:

```
0.0
```

In[5]:

```
entropy([1, 1, 1, 1, 1, 0, 0, 0])
```

Out[5]:

```
0.954434002924965
```

In[6]:

```
df_non_missing.Survived
```

Out[6]

```
183-element Vector{Int64}
1
1
0
1
1
1
1
0
1
0
0
1
0

and so on

1
1
1
1
1
1
0
1
0
1
1
1
```

In[7]:

```
entropy(df_non_missing.Survived)
```

Out[7]:

```
0.9127341558073343
```

In[8]:

```
entropy(df_men.Survived)
```

Out[8]:

```
0.9575534837147482
```

In[9]:

```
entropy(df_women.Survived)
```

Out[9]:

```
0.29461520565280713
```

In[10]:

```
entropy(df_children.Survived)
```

Out[10]:

```
0.4854607607459134
```

In[11]:

```
df_men
```

Out[11]:

Sno.	**Sex**	**Age**	**Survived**	**Pclass**
	String	Float64	Int64	Int64
1	male	54.0	0	1
2	male	34.0	1	2
3	male	28.0	1	1
4	male	19.0	0	1
5	male	65.0	0	1
6	male	45.0	0	1
7	male	25.0	0	3
8	male	46.0	0	1
9	male	71.0	0	1
10	male	23.0	1	1
11	male	21.0	0	1
12	male	47.0	0	1
13	male	24.0	0	1

Sno.	Sex	Age	Survived	Pclass
14	male	54.0	0	1
15	male	37.0	0	1
16	male	24.0	0	1
17	male	36.5	0	2

In[12]:

```
filter_1st_class_men = df_men.Pclass .== 1
df_men_first_class = df_men[filter_1st_class_men, :]
```

Out[12]:

	Sex	Age	Survived	Pclass
	String	Float64	Int64	Int64
1	male	54.0	0	1
2	male	28.0	1	1
3	male	19.0	0	1
4	male	65.0	0	1
5	male	45.0	0	1

In[13]:

```
entropy(df_men_first_class.Survived)
```

Out[13]:

```
0.9631672450918831
```

In[14]:

```
filter_other_class_men = .!filter_1st_class_men
df_men_other_class = df_men[filter_other_class_men, :]
```

Out[14]:

	Sex	Age	Survived	Pclass
	String	Float64	Int64	Int64
1	male	34.0	1	2

	Sex	Age	Survived	Pclass
2	male	25.0	0	3
3	male	36.5	0	2
4	male	36.0	0	2
5	male	32.0	1	3

In[15]:

```
entropy(df_men_other_class.Survived)
```

Out[15]:

```
0.863120568566631
```

In[16]:

```
women_survived = df_women.Survived
```

Out[16]:

```
77-element Vector{Int64}:
 1
 1
 1
 1
 1
 1
 1
 1
 1
 0
 1
 1
 1
 
 1
 0
 1
 1
 1
 1
 1
 1
 1
 1
 1
 1
```

In[17]:

```
counter(women_survived)
```

Out[17]:

```
Dict{Any, Any} with 2 entries:
  0 => 4
  1 => 73
```

In[19]:

```
highest_vote(counter(women_survived))
```

Out[19]:

```
1
```

In[20]:

```
counter(df_men_first_class.Survived)
```

Out[20]:

```
Dict{Any, Any} with 2 entries:
  0 => 49
  1 => 31
```

In[21]:

```
counter(df_children.Survived)
```

Out[21]:

```
Dict{Any, Any} with 2 entries:
  0 => 2
  1 => 17
```

In[22]:

```
size(df_children)
```

Out[22]:

```
(19, 2)
```

In[1]:

```
include("ml_lib.jl")
```

Out[1]:

```
entropy (generic function with 1 method)
```

In[2]:

```
using DataFrames
using CSV
```

In[3]:

```
df = DataFrame(CSV.File("data/titanic_feature_engineered.csv"))

# Get the non missing rows
df_non_missing = dropmissing(df)

# Get only adults
filter_adults = df_non_missing.Age .>= 18
df_adults = df_non_missing[filter_adults, :]

# Get only the men
filter_men = df_adults.Sex .== "male"
df_men = df_adults[filter_men, [:Sex, :Age, :Survived, :Pclass]]

# Get only the woman
filter_women = .!filter_men
df_women = df_adults[filter_women, [:Sex, :Age, :Survived, :Pclass]]

# Get only children
filter_children = .!filter_adults
df_children = df_non_missing[filter_children, [:Age, :Survived]]
```

Out[3]:

Sno	Age	Survived
	Float64	Int64
1	2.0	0
2	14.0	1
3	4.0	1
4	14.0	0
5	2.0	0

In[4]:

```
entropy(df_non_missing.Survived)
```

Out[4]:

```
0.9744414561311621
```

In[5]:

```
entropy(df_men.Survived)
```

Out[5]:

```
0.6739468651941155
```

In[6]:

```
entropy(df_women.Survived)
```

Out[6]:

```
0.7747817605858327
```

In[7]:

```
entropy(df_children.Survived)
```

Out[7]:

```
0.9954192904269324
```

In[8]:

```
df_men
```

Out[8]:

Sno	**Sex**	**Age**	**Survived**	**Pclass**
	String	Float64	Int64	Int64
1	male	22.0	0	3
2	male	35.0	0	3
3	male	54.0	0	1
4	male	20.0	0	3
5	male	39.0	0	3

In[28]:

```
counter(df_men.Survived)
```

Out[28]:

```
Dict{Any, Any} with 2 entries:
  0 => 325
  1 => 70
```

In[9]:

```
filter_1st_class_men = df_men.Pclass .== 1
df_men_first_class = df_men[filter_1st_class_men, :]
```

Out[9]:

Sno	Sex	Age	Survived	Pclass
	String	Float64	Int64	Int64
1	male	54.0	0	1
2	male	28.0	1	1
3	male	19.0	0	1
4	male	40.0	0	1
5	male	28.0	0	1

In[10]:

```
entropy(df_men_first_class.Survived)
```

Out[10]: ----0.9515388458648668----

In[30]:

```
counter(df_men_first_class.Survived)
```

Out[30]:

```
Dict{Any, Any} with 2 entries:
  0 => 61
  1 => 36
```

In[11]:

```
filter_other_class_men = .!filter_1st_class_men
df_men_other_class = df_men[filter_other_class_men, :]
```

Out[11]:

Sno	Sex	Age	Survived	Pclass
	String	Float64	Int64	Int64
1	male	22.0	0	3
2	male	35.0	0	3
3	male	20.0	0	3
4	male	39.0	0	3
5	male	35.0	0	2

In[12]:

```
entropy(df_men_other_class.Survived)
```

Out[12]:

```
0.512142399277117
```

In[32]:

```
counter(df_men_other_class.Survived)
```

Out[32]:

```
Dict{Any, Any} with 2 entries:
  0 => 264
  1 => 34
```

In[35]:

```
34 / size(df_men_other_class)[1]
```

Out[35]: ----0.11409395973154363----

In[36]:

```
size(df_men_other_class)
```

Out[36]:

```
(298, 4)
```

In[37]:

```
36 / size(df_men_first_class)[1]
```

Out[37]:

```
0.3711340206185567
```

In[13]:

```
women_survived = df_women.Survived
```

Out[13]:

```
206-element Vector{Int64}:
 1
 1
 1
 1
 1
 1
 0
 1
 0
 0
 0
 1
 0
 
 1
 1
 1
 1
 1
 1
 1
 1
 1
 0
 0
 1
```

In[14]:

```
counter(women_survived)
```

Out[14]:

```
Dict{Any, Any} with 2 entries:
  0 => 47
  1 => 159
```

In[15]:

```
highest_vote(counter(women_survived))
```

Out[15]:

```
1
```

In[16]:

```
counter(df_men_first_class.Survived)
```

Out[16]:

```
Dict{Any, Any} with 2 entries:
  0 => 61
  1 => 36
```

In[17]:

```
counter(df_children.Survived)
```

Out[17]:

```
Dict{Any, Any} with 2 entries:
  0 => 52
  1 => 61
```

In[18]:

```
size(df_children)
```

Out[18]:

```
(113, 2)
```

In[19]:

```
filter_1st_class_women = df_women.Pclass .== 1
```

Out[19]:

```
206-element BitVector:
 1
 0
 1
 0
 1
 0
 0
 0
 0
 0
 0
 0
 0
 
 1
 0
 1
 0
 0
 1
 0
 1
 0
 0
 0
 1
```

In[20]:

```
df_women_first_class = df_women[filter_1st_class_women, :]
```

Out[20]:

Sno	Sex	Age	Survived	Pclass
	String	Float64	Int64	Int64
1	female	38.0	1	1
2	female	35.0	1	1
3	female	58.0	1	1
4	female	49.0	1	1
5	female	38.0	1	1

In[21]:

```
entropy(df_women_first_class.Survived)
```

Out[21]:

```
0.173781322679433
```

In[22]:

```
filter_other_class_women = .! filter_1st_class_women
```

Out[22]:

```
206-element BitVector:
 0
 1
 0
 1
 0
 1
 1
 1
 1
 1
 1
 1
 1
 
 0
 1
 0
 1
 1
 0
 1
 0
 1
 1
 1
 0
```

In[23]:

```
df_women_other_class = df_women[filter_other_class_women, :]
```

Out[23]:

Sno	Sex	Age	Survived	Pclass
	String	Float64	Int64	Int64
1	female	26.0	1	3
2	female	27.0	1	3
3	female	55.0	1	2
4	female	31.0	0	3
5	female	38.0	1	3

In[24]:

```
entropy(df_women_other_class.Survived)
```

Out[24]:

```
0.9330252953592911
```

In[25]:

```
counter(df_women_first_class.Survived)
```

Out[25]:

```
Dict{Any, Any} with 2 entries:
  0 => 2
  1 => 75
```

In[26]:

```
counter(df_women_other_class.Survived)
```

Out[26]:

```
Dict{Any, Any} with 2 entries:
  0 => 45
  1 => 84
```

In[27]:

```
df_children
```

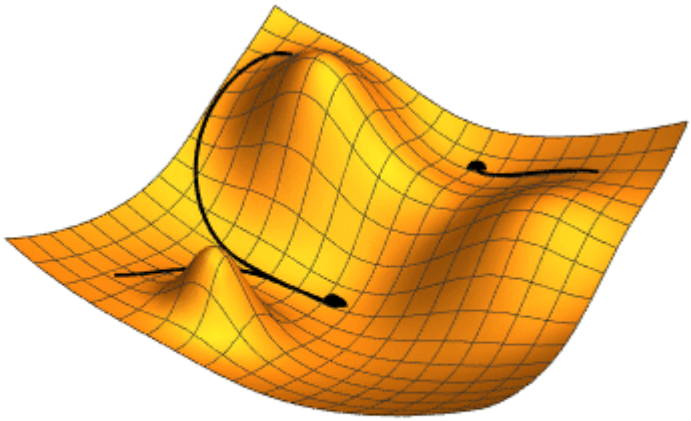
Out[27]:

Sno	Age	Survived
	Float64	Int64
1	2.0	0
2	14.0	1
3	4.0	1
4	14.0	0
5	2.0	0

48.4. Building a Decision Tree

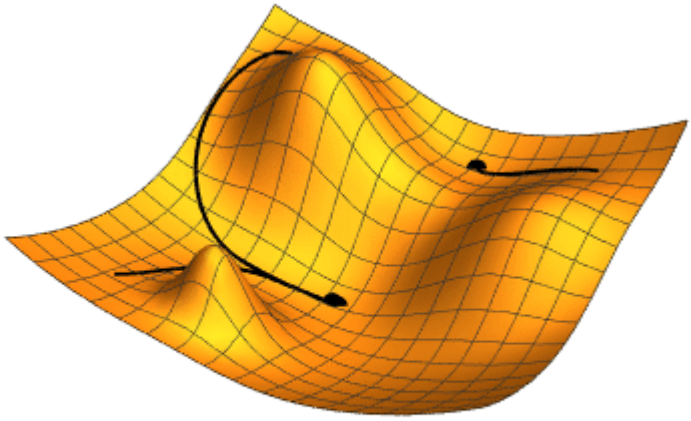
Get the Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/decision_tree_from_scratch_part_1.ipynb and https://gitlab.com/datascience-book/code/-/blob/master/decision_tree_from_scratch_part_2.ipynb

Chapter 49. Gradient Descent



Gradient descent seems to be one of the successful and miracle algorithm that seems to solve lot of machine learning problems. Let's say that there is a function $f(x)$, let's say you are told that it's been determined by two attributes a and b, you are given lot's of values of x and corresponding values of $f(x)$. So how would you find a and b?

Imagine you guess lots of combinations of a and b, then you plug those values into $f(x)$ for all values of x and you get the total square of error. You plot error vs a and b, you get something like this



Reader is encouraged to visit https://upload.wikimedia.org/wikipedia/commons/a/a3/Gradient_descent.gif to see gradient descent in action.

Now imagine you keep some balls at various points on the error graph, they will roll down the steepest gradient and descent to a place where a and b will produce the minimum deviation. That's the place where optimum a and b could be found.

Well if things are confusing, don't worry, let's see it in action.

49.1. Guessing Number With Gradient Descent

Get the Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/guessing_number_with_gradient_descent.ipynb

Video lecture for this section could be found here https://youtu.be/NNU7HgSDR-Q

First let's guesses a number with Gradient Descent.

49.1.1. Intuition

Let's say we have a number 42, we need to guess it. So in the piece of code below, we have `number` that's 42, `guess` , that's an initial random guess we make and `α`, which we call as step size. When yo descend down a hill, you don't spring like frog, you take small steps, similarly, we will change our `guess` in small steps to reach `number`.

```
number = 42
α = 0.01
guess = rand(0:100)
```

Output:

```
29
```

So in the case above, the computer has done a wild guess of 29, which is way off 42, now how to make the guess slightly more closer to 42. First we compute the error as shown below:

```
error = guess - number
```

Output:

```
-13
```

Then we recompute guess as shown below:

```
guess = guess - α * error
```

Output:

```
29.13
```

As you see, we don't do `guess= guess - error`, but we multply `error` with `α`. `α` is called step size or

learning constant.

So we see that we have moved guess slightly more closer to 42 by following the above steps. Now let's take 1000 steps.

```
for i in 1:1000
    error = guess - number
    guess = guess - α * error
end

guess
```

Output:

```
41.99944438604583
```

As you see, taking 1000 steps makes our guess very much closer to 42 which is our destiny. Even if we take 10,000 or 1,000,000 steps we won't over shoot 42, that's because, as we go near 42, `error` becomes almost zero thus change in guess keeps decreasing.

49.1.2. Guessing Number

Now we have got some intuition about guessing number with gradient descent, let's write some code to visualize it. The code below is same as we have seen in the previous section, but we have got variables `guesses` and `errors` which record our guesses and errors.

```
number = 42
α = 0.01
guess = rand(0:100)
guesses = []
errors = []

for i in 1:1000
    error = guess - number
    guess = guess - α * error
    push!(guesses, guess)
    push!(errors, error)
end
```

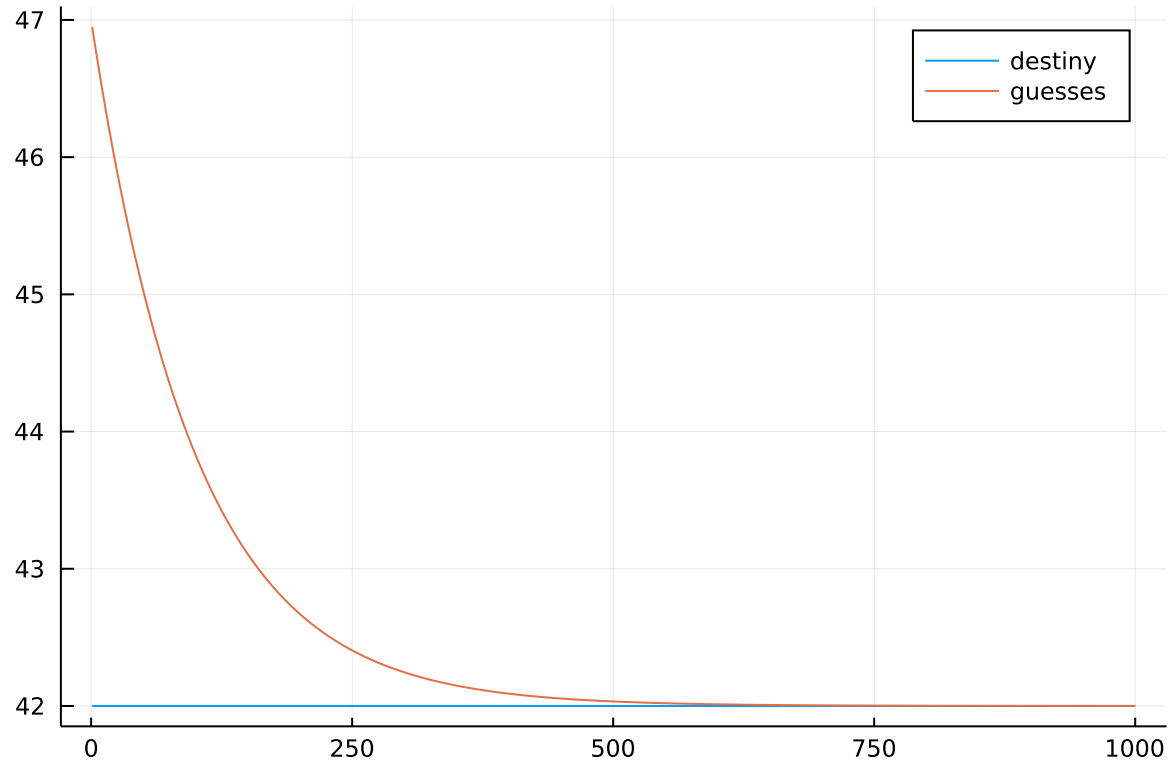
Now let's plot our `guesses` against our destiny `number`

```
using Plots

number_line = [number for i in 1:1000]

visual = plot!(1:1000, number_line, label="destiny")
plot!(visual, guesses, label="guesses")
```

Output:



As you can see initially, the random `guess` which is around 47, creases steeply, but at the same time our `error` to descends steeply thus limiting the rate of change of `guess`. Slowly the `guess` approaches number and error since it almost is `0` ensures that the guess never over shoots 42.

Exercise: Change the value of `α` to a value say 10 and see what happens? could you explain why?.

Exercise: Change `number` and rerun the above program. Does the plotted graph differ? How and why?

49.2. Linear Regression With Gradient Descent

Video lecture for this section could be found here https://youtu.be/9WaSusd8wBM

Now let's say you are given a bunch of points x_i's and y_i's, you plot it and looks like the dependent variable is linearly linked to feature, you suspect a prediction $\hat{y}$ will look like this:

$$\hat{y}_i = mx_i + c$$

So how to solve this by gradient descent assuming we have some wild random guess of m and c?

So let's first start by taking partial differentiation of $\hat{y}$ over m.

$$\frac{\text{del } \hat{y}_i}{\text{del } m} = \frac{\text{del}}{\text{del } m}(mx_i + c) = x_i$$

Now let's do the same, this time over c.

$$\frac{\text{del } \hat{y}_i}{\text{del } c} = \frac{\text{del}}{\text{del } c}(mx_i + c) = 1$$

We will use these results later.

Now say we have some wild guess of m and c, we plug it into $\hat{y}$ and we get an value for given x_i, this would be our error value:

$$e_i = \hat{y}_i - y_i$$

Now let's square it:

$$e_i^2 = \left(\hat{y}_i - y_i\right)^2$$

This is for one point x_i which gives prediction $\hat{y}_i$, for guessed value m and c, let's compute the total error for n given points:

$$e_{Tot}^2 = \sum_{i=0}^{n} \left(\hat{y}_i - y_i\right)^2$$

Since its better to get average error, let's divide e_{Tot}^2 by n:

$$e_{avg}^2 = \frac{1}{n}\sum_{i=0}^{n} \left(\hat{y}_i - y_i\right)^2$$

Let's call this equation E as shown below:

$$E = \frac{1}{n}\sum_{i=0}^{n} \left(\hat{y}_i - y_i\right)^2$$

Now we differentiate E partially with m:

$$\frac{\text{del } E}{\text{del } m} = \frac{1}{n}\sum_{i=0}^{n} \frac{\text{del}}{\text{del } m}\left(\hat{y}_i - y_i\right)^2$$

which gives:

$$\frac{\text{del } E}{\text{del } m} = \frac{2}{n}\sum_{i=0}^{n} \left(\hat{y}_i - y_i\right) \cdot \frac{\text{del}}{\text{del } m}\left(\hat{y}_i - y_i\right)$$

Since y_i can be treated as constant, we can remove it since:

$$\frac{\text{del } y_i}{\text{del } m} = 0$$

So we get:

$$\frac{\text{del } E}{\text{del } m} = \frac{2}{n}\sum_{i=0}^{n} \left(\hat{y}_i - y_i\right) \cdot \frac{\text{del}}{\text{del } m}\left(\hat{y}_i\right)$$

Now plugging in $\frac{\text{del}}{\text{del } m}\left(\hat{y}_i\right)$ we get:

$$\frac{\text{del } E}{\text{del } m} = \frac{2}{n}\sum_{i=0}^{n}\left(\hat{y}_i - y_i\right)\cdot x_i$$

Now we can update m taking a step in the direction and magnitude pointed by gradient $\frac{\text{del } E}{\text{del } m}$, before subtracting from m, we multiply it with constant α as shown:

$$m := m - \alpha\frac{\text{del } E}{\text{del } m}$$

Similarly $\frac{\text{del } E}{\text{del } c}$ gives:

$$\frac{\text{del } E}{\text{del } c} = \frac{2}{n}\sum_{i=0}^{n}\left(\hat{y}_i - y_i\right)$$

We can obtain new value of c as follows:

$$c := c - \alpha\frac{\text{del } E}{\text{del } c}$$

So if we repeat these steps of finding the error gradient, multiply with a small constant α and minus the value from current value on and on, we will naturally come down to a point where m and c gives minimum value of error, thus guessing almost the right values of m and c.

49.2.1. Code

Get the Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/linear_regression_with_gradient_descent.ipynb

In the last section you saw about the mathematics of solving a simple linear equation with gradient descent, now let's see it in code. First we have a function that describes a equation of line:

```
line(m, x, c) = m * x + c
```

Let's test it out:

```
line(2, 5, 3)
```

Output:

```
13
```

Seems to work!. Now let's generate sample treating data for us. We describe a function `y(x)` as shown below:

```
y(x) = line(2, x, 3)
```

So when we run gradient descent, we expect our values of `m` and `c` to converge upon 2 and 3 respectively.

Let's test the function:

```
y(5)
```

Output:

```
13
```

Seems to work!

Now let's say we take sample `X` to be 1 to 10 as shown:

```
X = 1:10
```

Output:

```
1:10
```

We generate sample `Y` for training as shown below:

```
Y = y.(X)
```

Output:

```
10-element Vector{Int64}:
  5
  7
  9
 11
 13
 15
 17
 19
 21
 23
```

Now for given training data `X` and `Y`, for a guessed `m` and `c` we need to find errors, for that we write a function as shown:

```
function errors(X, Y, m, c)
    predictions = [line(m, x, c) for x in X]
    predictions - Y
end
```

Now let's test it out, we aasume `m` is 4 and `c` is 5:

```
m = 4
c = 5
α = 0.001
```

```
errors(X, Y, m, c)
```

```
10-element Vector{Int64}:
  4
  6
  8
 10
 12
 14
 16
 18
 20
 22
```

Seems to work!

Now let's load the cont of data points we have into a a variable `n` as shown:

```
n = length(Y)
```

Output:

```
10
```

If you remember we had assumed `m` as 4 and `c` as 5 before. Now let's update `m` using the gradient:

```
m = m - (2 / n) * α * sum(errors(X, Y, m, c) .* X)
```

Output:

```
3.824
```

So we see its reducing, that is moving towards 2. Similar let's update for `c`:

```
m = 4
c = 5

c = c - (2 / n) * α * sum(errors(X, Y, m, c))
```

Output:

```
4.974
```

We see it too is climbing down from 5, that is traveling towards 3.824

Now with the new values of `m` and `c` let's update `m` once again:

```
m = 3.824
c = 4.974

m = m - (2 / n) * α * sum(errors(X, Y, m, c) .* X)
```

Output:

```
3.661838
```

It decreases further, near to 2, our expected `m`.

And now for c

```
m = 3.824
c = 4.974

c = c - (2 / n) * α * sum(errors(X, Y, m, c))
```

Output:

```
4.949988
```

It decreases further! So we are going somewhere!!!

Now let's assume `m` as 5 and `c` as 0, now let's apply this gradient descent algorithm and let's take a 1000 steps:

```
m = 5
c = 0
α = 0.01

for i in 1:1000
    Δ = errors(X, Y, m, c)
    m = m - (2 / n) * α * sum(Δ .* X)
    c = c - (2 / n) * α * sum(Δ)
end

(m, c)
```

Output

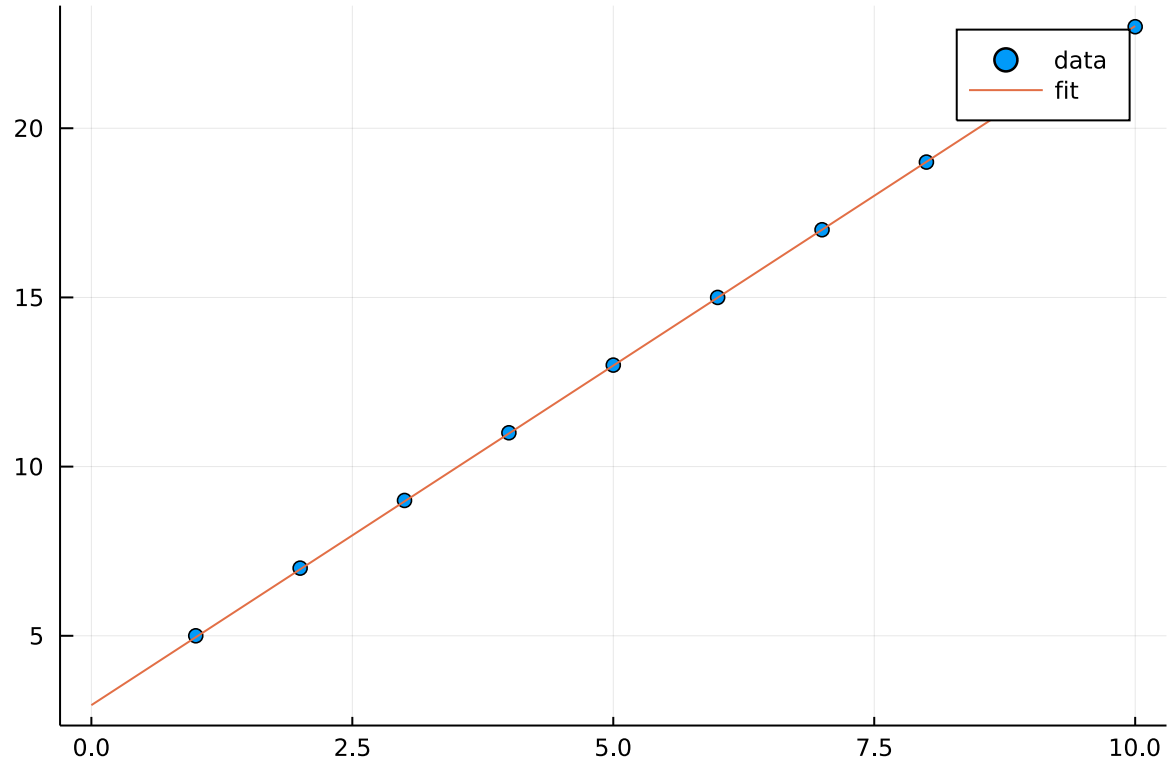
```
(2.0071802776754826, 2.9500121822629732)
```

Wow! `m` is almost 2 and `c` is almost 3, now let's plot our data points and our guessed line:

```
using Plots

graph = scatter(X, Y, label = "data")
points = 0:0.1:10
plot!(graph, points, [line(m, x, c) for x in points], label = "fit")
```

Output:



A very tight fit, even though its not 100% perfect. Good job!

49.3. Generalizing Linear Regression With Gradient Descent

Now since you have done gradient descent for $y = mx + c$, let's generalize it. Say suppose you have a table as shown below:

Table 1. Feature List

Feature 1	...	Feature j	...	Feature m	Prediction
$x_{1,\ 1}$	...	$x_{1,\ j}$	...	$x_{1,\ m}$	y_1
...	...	...	...	...	...
$x_{i,\ 1}$	...	$x_{i,\ j}$	...	$x_{i,\ m}$	y_i
...	...	...	...	...	...
$x_{n,\ 1}$	...	$x_{n,\ j}$	...	$x_{n,\ m}$	y_n

There are m features, and n rows or data points, so you think this relation between features and prediction y_i might be related like this:

$$h = c + \sum_{j=1}^{m} a_j x_j$$

That is if a_j is approximated to the right amounts than you can get a good prediction.

h is used because, h is the first letter in the word hypothesis.

So for each ith row the prediction would be like:

$$\hat{y}_i = c + \sum_{j=1}^{m} a_j x_{i,\ j}$$

Now doing a partial differentiation of y_i with respect to a_j we get:

$$\frac{\text{del } \hat{y}_i}{\text{del } aj} = x_{i,\ j}$$

We will use the above result later.

Similarly doing partial differentiation of y_i with respect to c we get:

$$\frac{\text{del } \hat{y}_i}{\text{del } c} = 1$$

We will use the above result later.

Now the average square of errors for n data points would be:

$$E = \frac{1}{n} \sum_{i=1}^{n} \left(\hat{y}_i - y_i \right)^2$$

So doing partial differentiation of E with respect to a_j we get:

$$\frac{\text{del } E}{\text{del } a_j} = \frac{2}{n} \sum_{i=1}^{n} \left(\hat{y}_i - y_i \right) \cdot \frac{\text{del } \hat{y}_i}{\text{del } a_j}$$

$$\frac{\text{del } E}{\text{del } a_j} = \frac{2}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right) \cdot x_{i,\ j}$$

Similarly doing partial differentiation of E with respect to c we get:

$$\frac{\text{del } E}{\text{del } c} = \frac{2}{n}\sum_{i=1}^{n}\left(\hat{y}_i - y_i\right)$$

So once the above procedures are done in code, we can obtain new value of a_j as:

$$a_j := a_j - \alpha\frac{\text{del } E}{\text{del } a_j}$$

Where α is the step size.

Similarly we can get the new value of c as shown:

$$c := c - \alpha\frac{\text{del } E}{\text{del } c}$$

Chapter 50. Hot and Cold Learning

Chapter 51. K Means Clustering

Video lecture for this section could be found here https://youtu.be/468U1e4ITjI

K Means Clustering is an unsupervised learning. That is if you see in this book you need to some for of label to data that is in the Titanic case you need to say if the person has survived or not, in Iris flower classification you need to say what type of Iris the flower is. In K Means, that is not the case. It's used in cases where you do not really know how to classify the data you have.

Let's say you have a bunch of data points, you throw it to K Means and tell it to classify in 10 ways, and it will do then it's on you to make sense of it.

51.1. Intuition

So let's try to understand it. So you have got six points as shown below. Assume that they are on cartesian plane. You as a human can say intuitively that you can bunch it into two, but how does the computer do it. It does not have the millions of years of evolutionary advantage. You inherit a crocodile's brain when your computers processors was made, it almost inherited no learning.

So this is how you do it, you randomly assign two points (in this case I want to classify it two ways, that is K = 2), let's say one get classified as red group and one gets classified as blue group. So we place a red cross on a random point, and a blue cross on another random point. Let's say they are the mid points of red and blue group as shown below:

Now all points near red midpoints become part of the red group and all points below the blue mid point becomes part of the blue group as shown below.

Now we recenter our midpoint, as its no longer the mid point, so the red and blue crosses becomes the mid point of the group again as shown below.

Now if we once again try to color the points, the coloring wont change, as the blue dots at the right are near to the blue mid point, and red dots at the left are near the red mid point. Now we can conclude the clustering is complete.

Now take another case, we initially choose two random midpoint, the red mid point is the lowest left dot and blue mid point is next to it. So when we color the dots blue dominates and we as humans could intuitively say that this clustering seems out of place. So how to mitigate this?

Once again recalculate the mid points, the red midpoint stays put, but the blue one is pulled far to right by the weight of three dots to the right as shown below.

Now once again recolor, so the dots near the red mid point becomes red, and the ones near blue is blue as shown. Magically this recalculating and recoloring seems to fix the issues as shown below.

One again reposition the mid points as shown below and you are done. This is how K Means clustering is done.

As a human you can cluster points from 1D to 3D space, beyond that it will be difficult for human. Plus if the points are in thousands and millions, it becomes really difficult. So let's write Julia code for it.

51.2. Writing it in Julia

Get the Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/k_means_clustering.ipynb

Let's write the K Means Clustering in Julia, first we import vector library which we developed in the vectors section

```
include("lib/vect_lib.jl")
```

You can find that library here https://gitlab.com/datascience-book/code/-/blob/master/vectors.ipynb

Now of the main thing in K Means is you must be able to associate a point (here in Julia we are representing as a vector), to a mid point. So in the function below, we take in all the mid points, and a vector, we see to which mid point the vector is nearest, and associate the vector with that mid point.

```
function nearest(midpoints, vector)
    k = length(midpoints)

    index = 1
    output = Dict()
    Δd = distance(midpoints[1], vector)

    for midpoint in midpoints
        new_distance = distance(midpoint, vector)

        if Δd >= new_distance
            output["distance"] = Δd
            output["midpoint"] = midpoint
            output["midpoint_index"] = index
        end

        index += 1
    end

    output
end
```

Let's execute the above function, we give two mid points `[1, 1]` and `[5, 5]`, and we give a vector or point in question as `[6, 6]`

```
nearest(
    [ [1, 1], [5, 5] ],
    [6, 6]
)
```

Naturally, `[6, 6]` is near `[5, 5]` and hence we get an output as shown. The output we get is in the form of dictionary and it contains three attributes, they are:

```
Dict{Any, Any} with 3 entries:
  "midpoint"       => [5, 5]
  "distance"       => 7.07107
  "midpoint_index" => 2
```

- `midpoint` - Which says which mid point is nearest to the vector in question
- `distance` - Which says whats the distance from the vector in question to the mid point
- `midpoint_index` - Which says at what indes the mid point is in the passed array.

When we give two midpoints `[5, 5]` and `[7, 7]` which are equidistance from `[6, 6]`, out algorithm picks the last one as shown:

```
nearest(
    [ [1, 1], [5, 5], [7, 7] ],
    [6, 6]
)
```

Output:

```
Dict{Any, Any} with 3 entries:
  "midpoint"       => [7, 7]
  "distance"       => 7.07107
  "midpoint_index" => 3
```

Similarly we check which midpoint is closest to `[0, 0]` below:

```
nearest(
    [ [1, 1], [5, 5], [7, 7] ],
    [0, 0]
)
```

Output:

```
Dict{Any, Any} with 3 entries:
  "midpoint"       => [1, 1]
  "distance"       => 1.41421
  "midpoint_index" => 1
```

Now we check which is closest to `[3, 4]` below:

```
nearest(
    [ [1, 1], [5, 5], [7, 7] ],
    [3, 4]
)
```

Output:

```
Dict{Any, Any} with 3 entries:
  "midpoint"       => [5, 5]
  "distance"       => 3.60555
  "midpoint_index" => 2
```

Now let's tackle he heart of our problem, given a bunch of mid points and bunch of vectors, we need say to which mid point the vector could be clustered. Thats been tackled in the code below.

```
function cluster(midpoints, vectors)
    k = length(midpoints)
    clusters = [[] for i in 1:k]

    for vector in vectors
        nearest_info = nearest(midpoints, vector)
        index = nearest_info["midpoint_index"]
        push!(clusters[index], vector)
    end

    clusters
end
```

Let's go through the above function in detail. So let's say we start with a empty function:

```
function cluster(midpoints, vectors)
end
```

From `k` is nothing but number of midpoints:

```
function cluster(midpoints, vectors)
    k = length(midpoints)
end
```

Now we create vector of `k` empty vectors for clustering:

```
function cluster(midpoints, vectors)
    k = length(midpoints)
    clusters = [[] for i in 1:k]
end
```

Now for each vector:

```
function cluster(midpoints, vectors)
    k = length(midpoints)
    clusters = [[] for i in 1:k]

    for vector in vectors
    end
end
```

We get the nearest midpoint:

```
function cluster(midpoints, vectors)
    k = length(midpoints)
    clusters = [[] for i in 1:k]

    for vector in vectors
        nearest_info = nearest(midpoints, vector)
    end
end
```

We know the nearest mid point has the midpoint_index which tells to which mid point the vector is nearest, so we get that into a variable named `index`:

```
function cluster(midpoints, vectors)
    k = length(midpoints)
    clusters = [[] for i in 1:k]

    for vector in vectors
        nearest_info = nearest(midpoints, vector)
        index = nearest_info["midpoint_index"]
    end
end
```

Now we add the vector to the cluster at that particular index. Finally we return the clusters out.

```
function cluster(midpoints, vectors)
    k = length(midpoints)
    clusters = [[] for i in 1:k]

    for vector in vectors
        nearest_info = nearest(midpoints, vector)
        index = nearest_info["midpoint_index"]
        push!(clusters[index], vector)
    end

    clusters
end
```

Let's test the function:

```
cluster(
    [ [0, 0], [100, 100] ],
    [ [0,0], [102, 20], [5, 5], [105, 105] ]
)
```

Output:

```
2-element Vector{Vector{Any}}:
 [[0, 0], [5, 5]]
 [[102, 20], [105, 105]]
```

Seems to wok!

Now in `vect_lib.jl` we already have coded a `midpoint` function, let's test it out.

```
midpoint([[0, 0], [5, 5]])
```

```
2-element Vector{Float64}:
 2.5
 2.5
```

```
midpoint([[102, 20], [105, 105]])
```

Output:

```
2-element Vector{Float64}:
 103.5
  62.5
```

Seems to work!

All we need to do now is to code our own K Means Clustering function. So in the function below takes in vector of vectors into variable named `vectors`, and it takes the values of `k`, and it groups the vectors into `k` clusters.

```
function k_means_clustering(vectors, k = 2)
    vect_length = length(vectors)
    centers = []
    clusters = nothing

    for i in 1:k ①
        push!(centers, vectors[rand(1:vect_length)])
    end

    for i in 1:ceil(vect_length / k)
        clusters = cluster(centers, vectors)  ②
        centers = []

        for vect_array in clusters ③
            center = midpoint(vect_array)
            push!(centers, center)
        end
    end

    clusters
end
```

① Pick K random midpoints / centers

② Cluster according to centers

③ Recenter midpoints according to new clusters

All we do in the code is first pick out `k` centers or midpoints, then for number of vectors `vect_length` divided by `k` times we find clusters and re center again. This clusters the vectors.

So let's try it out:

```
k_means_clustering([ [0,0], [102, 20], [5, 5], [105, 105], [3, 4], [67, 82], [50, 50]
])
```

Output:

```
2-element Vector{Vector{Any}}:
 [[0, 0], [5, 5], [3, 4]]
 [[102, 20], [105, 105], [67, 82], [50, 50]]
```

Seems to work.

Now let's try to cluster our vector into 3 clusters:

```
k_means_clustering([ [0,0], [102, 20], [5, 5], [105, 105], [3, 4], [67, 82], [50, 50]
], 3)
```

Output:

```
3-element Vector{Vector{Any}}:
 [[102, 20]]
 [[0, 0], [5, 5], [3, 4]]
 [[105, 105], [67, 82], [50, 50]]
```

Seems to work!

Sometimes this code breaks if the numbers of points is very less compared to k, to mitigate that use large number of points, some times just by running the function again with same data fixes it. I think the problem is with choosing same centers more than once.

Chapter 52. Naive Bayes For Text Classification

Video lecture for this section could be found here https://youtu.be/oHaeVuIhYoc

Get Jupyter notebook for this section here https://gitlab.com/datascience-book/code/-/blob/master/naive_bayes.ipynb

Naive Baye's is used for text classification, take a look at the labeled data set below, we have got text at left and label at the right.

Text	Label
A great game	Sport
The election was over	Politics
Very clean match	Sport
A clean but forgettable game	Sport
It was a close election	Politics

Armed with the data above, we will apply a thing called Baye's Theorem[12] to see if unknown text is either related to Politics or Sport.

We can represent this Data set above using this dataframe

```
df = DataFrame(
    text = [
        "A great game",
        "The election was over",
        "Very clean match",
        "A clean but forgettable game",
        "It was a close election"
    ],

    label = [
        "Sport",
        "Politics",
        "Sport",
        "Sport",
        "Politics"
    ]
)
```

Now let's see how to manually calculate the probability of words in each label. First, let's take the data only for politics

Text	Label
The election was over	Politics
It was a close election	Politics

Now it contains total of 9 words. Out of these 9 words two are `election`. So the probability of finding `election` in the label `Politics` is given as

$$P(\text{election}|\text{Politics}) = \frac{2}{9}$$

Similarly when we take the data for label `Sport`:

Text	Label
A great game	Sport
Very clean match	Sport
A clean but forgettable game	Sport

We find there are 11 words, and 2 of them are `game`. So probability of finding the word `game` in sport is:

$$P(\text{game}|\text{Sport}) = \frac{2}{11}$$

Now let's take a unseen string `a very close game`, we need to calculate if this belongs to label `Sport` or `Politics`.

To determine if this string belong to post, we use this formula, where we take each word, find the probability of the word in `Sport`, then multiply all of then together as shown:

$$P(\text{a very close game}|\text{Sport}) = P(a|Sport) \times P(very|\text{Sport}) \times P(close|\text{Sport}) \times P(game|\text{Sport})$$

But there is a problem here. There is no word named `close` in `Sport` data set, hence the probability will be zero:

$$P(\text{close}|\text{Sport}) = \frac{0}{11}$$

To mitigate this, we employ a technique called additive smoothing [13]. Here we are defining a function called additive smoothing as shown below:

$$\text{additive smoothing}\left(\frac{Nr}{Dr}\right) = \> ; \frac{Nr + \alpha}{N + \alpha Dr}$$

So the numerator Nr is 0 for us.

The denominator Dr is dimension or number of words in the given label, for this case the label is `Sport` and the total number of words is 11.

N is number of possible / unique words in the entire data set, and if you count them you will get 14 unique words.

α is smoothing constant, we have chosen 1

So plugging these values in the equation

$Nr = 0$, $Dr = 11$, $N = 14$, $\alpha = 1$

we get

$$P(\text{close}|\text{Sport}) = \frac{0}{11} = \frac{0+\alpha}{N+\alpha d} = \frac{0+1}{14+11\times 1} = \frac{1}{25} = 0.04$$

In general we can express the formula for checkingthe probility of given set of words in a label as shown:

$$P(\text{words}|\text{label}) = \prod_{\text{each word}} \text{additive smoothing}(P(\text{word}|\text{label}))$$

If you look at the code, I would have defined additive smoothing function as follows:

```
"""
Performs additive smoothing as mentioned in
https://en.wikipedia.org/wiki/Additive_smoothing

```julia
additive_smoothing(numerator, denominator, N = 1, α = 0)
```
N is number of trials

α is smoothing paramater
"""
function additive_smoothing(numerator, denominator, N = 0, α = 1)
    (numerator + α) / (N + denominator * α)
end
```

In the above code just concentrate on

```
function additive_smoothing(numerator, denominator, N = 0, α = 1)
    (numerator + α) / (N + denominator * α)
end
```

This is the additive smoothing code.

Now let's define a function to calculate the probability of word occurring in a function:

```
function word_probability(word, classification, data_frame, config = Dict())
    α = get(config, "α", 1)
    text_column = get(config, "text_column", "text")
    label_column = get(config, "label_column", "label")
    df = data_frame

    all_text = lowercase(join(df[:, text_column], " "))
    words = split(all_text)
    N = length(Set(words))

    filter = df[:, label_column] .== classification
    df_filtered = df[filter, :]
    filtered_text = lowercase(join(df_filtered[:, text_column], " "))

    word_counts = counter(split(filtered_text))

    word_count = get(word_counts, word, 0)

    additive_smoothing(word_count, length(split(filtered_text)), N, α)
end
```

The above function calculate the probability of word occurring in a given label. This is how it works.

We have a function named `word_probability`

```
function word_probability(word, classification, data_frame, config = Dict())
end
```

the first argument is `word`, which accepts the word who's probability should be determined, the second argument is `classification`, that is the label, we will be determining the probability of `word` occurring in `classification`. The third argument `data_frame` accepts the data frame that has the text and classification labels. The fourth is a argument called `config`, to which a dictionary should be passed.

The `config` could have the following keys. `α`, is the smoothing constant that we need to pass to the `additive_smoothing` function. `text_column` is the key that holds the name of the text column that needs to be considered. `label_column` is the key the holds the name of the label or classification column.

In these lines:

```
α = get(config, "α", 1)
text_column = get(config, "text_column", "text")
label_column = get(config, "label_column", "label")
```

We set default value for config keys, if nothing is in `α` it ddefaults to `1`, if nothing is in `text_column`, it

defaults to `text`, if nothing is in `label_column` it defaults to `label`. These are captured in varables `α`, `text_column` and `label_column`

I assign `data_frame` to a variable `df` in this line `df = data_frame` for convenience.

In these lines (shown below) in `word_probability` function, we take all text in the data frame, and get the count of unique words and assign it to variable named `N`. This `N` will be used in `additive_smoothing` function soon.

```
all_text = lowercase(join(df[:, text_column], " "))
words = split(all_text)
N = length(Set(words))
```

Now if the classification or label is `Sport` we need to take just text in sport, for that we need to filter it out, that is take the text rows that belong only to `classification` for that we write a `filter` as shown below:

```
filter = df[:, label_column] .== classification
```

Use the filter to filter the dataframe:

```
df_filtered = df[filter, :]
```

Then take out the filtered text:

```
filtered_text = lowercase(join(df_filtered[:, text_column], " "))
```

Now we count the words in that classification using the following line:

```
word_counts = counter(split(filtered_text))
```

For this we use counter function which we have it in our library `ml_lib.jl`[14].

Next we see if a word is present, if yes get its count, or else we get it as zero as shown below:

```
word_count = get(word_counts, word, 0)
```

Finally we pass all the value to `additive_smoothing` function which get's returned out:

```
additive_smoothing(word_count, length(split(filtered_text)), N, α)
```

Now you can try what's the word probability of of say `game` in label `Sport`:

```
word_probability("game", "Sport", df, Dict("α" => 1))
```

Output

```
0.12
```

Now string probability is the multiplication of word probabilities, so we write a function as shown:

```
function string_probability(string, classification, data_frame, config = Dict())
    α = get(config, "α", 1)
    text_column = get(config, "text_column", "text")
    label_column = get(config, "label_column", "label")

    config_with_defaults = Dict("α" => α,
                                "text_column" => text_column,
                                "label_column" => label_column)

    probability = 1

    string = replace(string, r"\W" => " ")
    words = split(lowercase(string))

    for word in words
        probability *= word_probability(word, classification, data_frame,
config_with_defaults)
    end

    probability
end
```

We get a `string` and a `classification` for which its probability that should be determined. We split the sting to words after removing punctuation and converting it to lower case in these lines:

```
string = replace(string, r"\W" => " ")
words = split(lowercase(string))
```

Next for each word we calculate the probability of it in the `classification` and multiply it all in these lines

```
for word in words
    probability *= word_probability(word, classification, data_frame, config)
end
```

And finally we return the `probability`.

Now let's se whats the probability of `string = "a very close game"` in `"Sport"`

```
string_probability(string, "Sport", df, Dict("α" => 1))
```

Output:

```
4.6079999999999999e-5
```

Now let's see its probability of being in label `"Politics"`:

```
string_probability(string, "Politics", df, Dict("α" => 1))
```

Output

```
1.4293831139825827e-5
```

So `string = "a very close game"` has is more probable to be in `Sport` than `Politics`.

One may write a function to loop through the classifications in dataframe for a given string, calculate the probability of string in for a classification, and take the label with max probability as label or classification for that string, such kind of function is written below:

```
function label_text(text, data_frame, config)
    α = get(config, "α", 1)
    text_column = get(config, "text_column", "text")
    label_column = get(config, "label_column", "label")

    config_with_defaults = Dict("α" => α,
                                "text_column" => text_column,
                                "label_column" => label_column)
    df = data_frame

    max_probability = 0
    label = nothing
    classifications = Set(df[:, label_column])

    for classification in classifications
        probability = string_probability(text, classification, df,
config_with_defaults)

        if probability > max_probability
            max_probability = probability
            label = classification
        end
    end

    label
end
```

Now let's test it for `"a very close game"`

```
string = "a very close game"
label_text(string, df, Dict("α" => 1))
```

And we get `Sport` as its classification or label. For `"I am a politician who won election"`, we get `Politics` as classification:

```
string = "I am a politician who won election"
label_text(string, df, Dict("α" => 1))
```

[12] https://en.wikipedia.org/wiki/Bayes%27_theorem

[13] https://en.wikipedia.org/wiki/Additive_smoothing

[14] https://gitlab.com/datascience-book/code/-/blob/master/lib/ml_lib.jl

Neural Networks

Chapter 53. Back propagation

Bibliography

- Think Julia - https://benlauwens.github.io/ThinkJulia.jl/latest/book.html
- Artificial Intelligence, a Modern Approach
- Machine Leaning in Action
- [mml] Mathematics for Machine Learning - https://mml-book.github.io/
- Data Science from Scratch
- Real World Machine Learning
- Super Intelligence
- The Joy of X
- Introduction to Probability for Data Science - https://probability4datascience.com/
- Stat Quest - https://www.youtube.com/channel/UCtYLUTtgS3k1Fg4y5tAhLbw
- Right2Trick - https://www.youtube.com/c/Right2Trick

Thank You!

Data Science with Julia