Download this presentation from
http://connettiva.eu/rubyday
Plus links to elixir and phoenix resources
Plus three HOWTOs to install erlang, elixir, phoenix.

Phoenix demo app at
https://github.com/pmontrasio/phoenix-demo-app

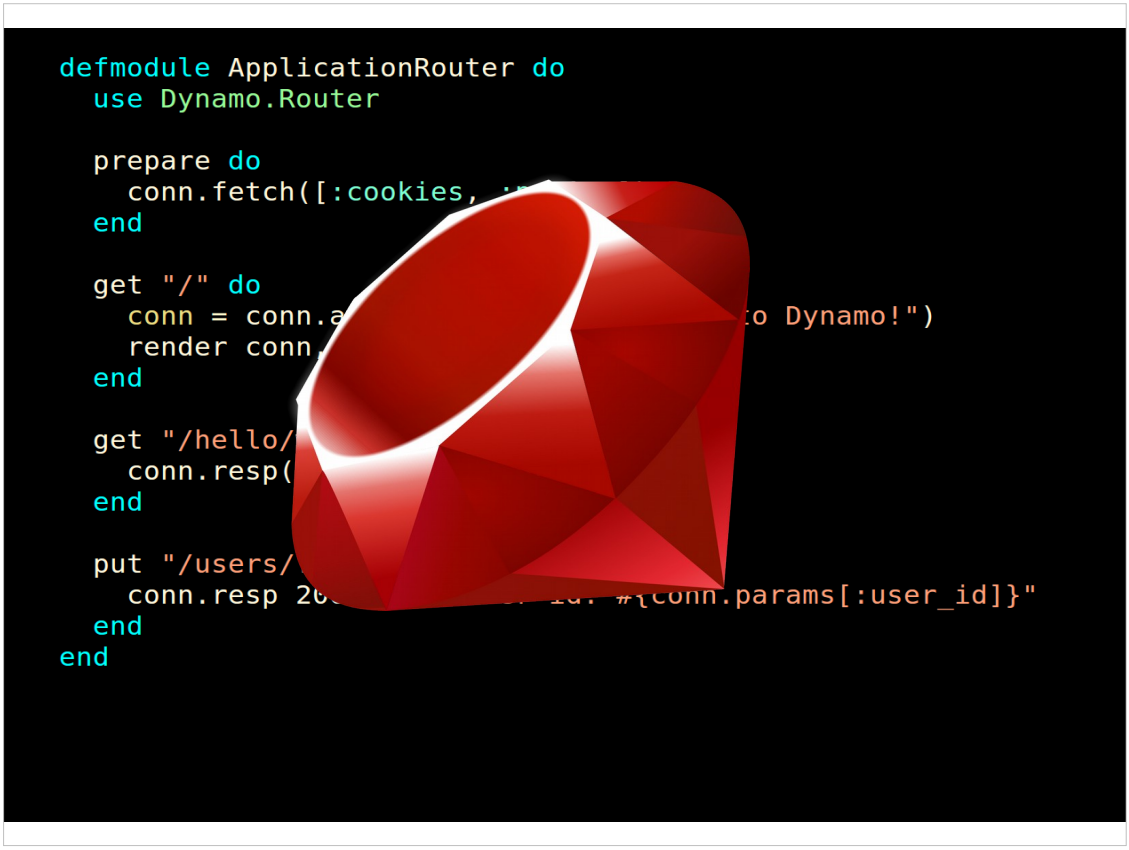
http://commons.wikimedia.org/wiki/File:Pomegranate_Seeds.JPG
Public domain

This speech is about things that look like other things but are not those things.

They look like rubies but they are not rubies. They are pomegrate seeds.

```
defmodule ApplicationRouter do
  use Dynamo.Router

  prepare do
    conn.fetch([:cookies, :params])
  end

  get "/" do
    conn = conn.assign(:title, "Welcome to Dynamo!")
    render conn, "index.html"
  end

  get "/hello/world" do
    conn.resp(200, "Hello world")
  end

  put "/users/:user_id" do
    conn.resp 200, "Got user id: #{conn.params[:user_id]}"
  end
end
```

This looks like Ruby.
It might be Sinatra or some other lightweight framework.



Is this Ruby?

http://commons.wikimedia.org/wiki/File:Ruby_logo.png
Yukihiro Matsumoto, Creative Commons Attribution-Share Alike 2.5 Generic

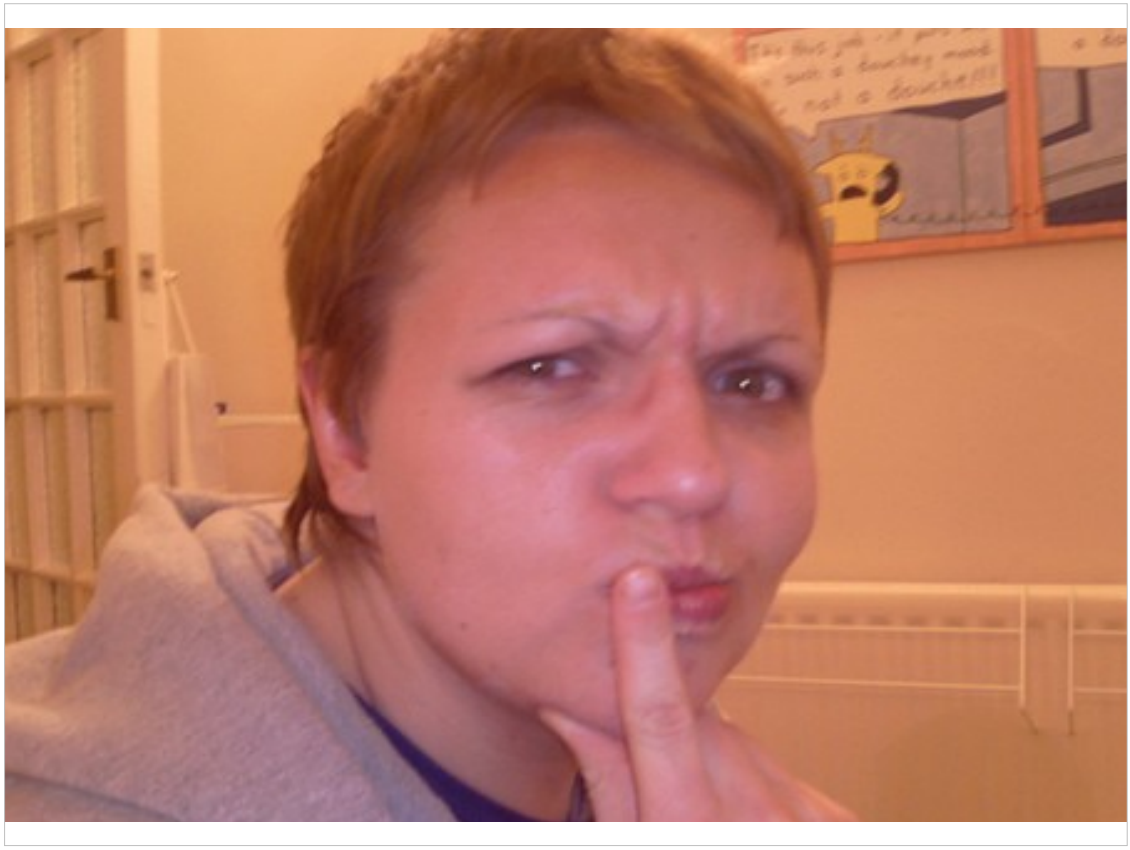
No, this is Elixir

Logo from http://elixir-lang.org/

Elixir is a language built on the top of the Erlang VM

Logo from http://elixir-lang.org/
© Plataformatec

Logo from http://commons.wikimedia.org/wiki/File:Erlang_logo.png
Public domain



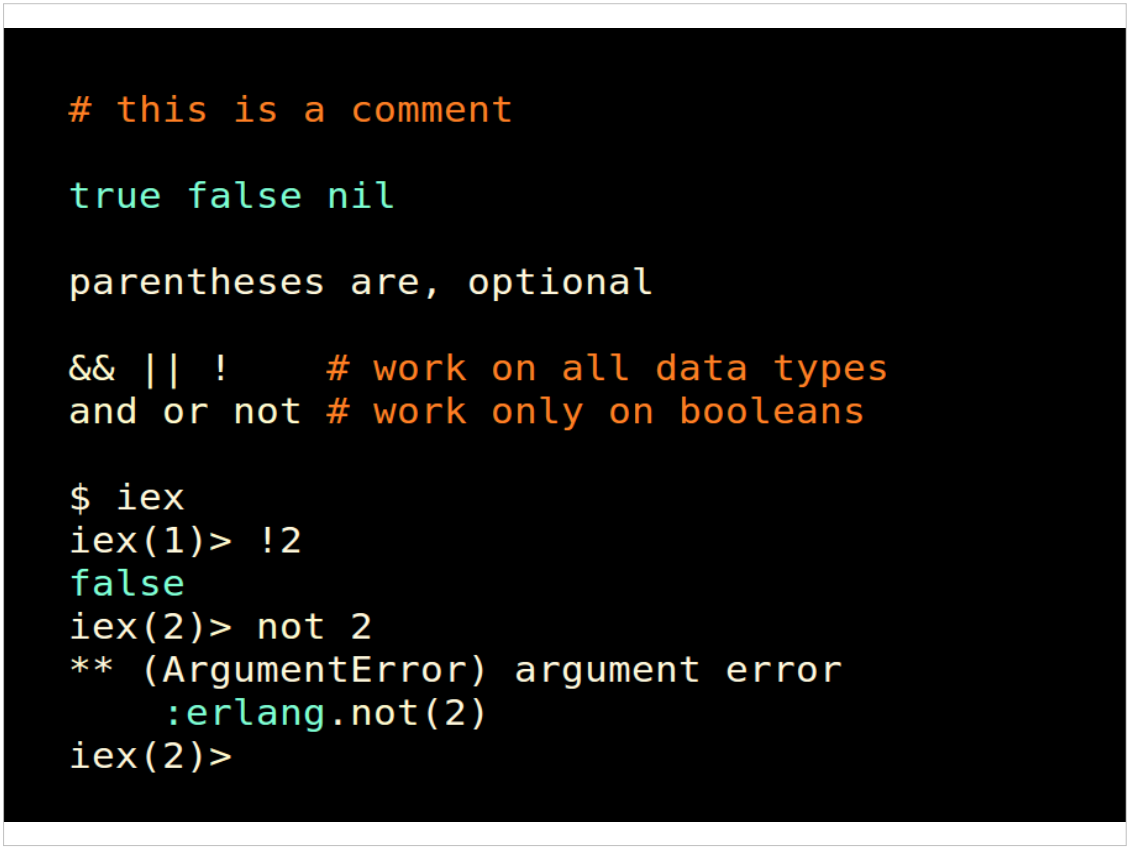
So we have a language that looks like Ruby, but actually is Elixir, which deep inside is Erlang. Puzzled?

https://www.flickr.com/photos/thesheriff/112137491/
Andrew Mitchell

Probably not :-)

Logo from http://elixir-lang.org/
© Plataformatec

Logo fom http://commons.wikimedia.org/wiki/File:Erlang_logo.png
Public domain



Let's start with some simple syntactical elements.
Comments are like in any other Unix based scripting language.
Parentheses are optional.
No end of statement terminator.
The C-like logical operators are what we expect.
They work on any data type.
The English-worded logical operators work only on booleans.
There is a interactive interpreter, iex, which is like irb.

```
if condition do
  ...
end

unless condition do
  ...
end

# one liners
if condition, do: ...
unless condition, do: ...

iex(2)> if !nil, do: "Look ma, one line!"
"Look ma, one line!"
```

Conditionals are like the Ruby ones, with the exception of the do after the condition.
One liners are quite different.
No postfix notation.

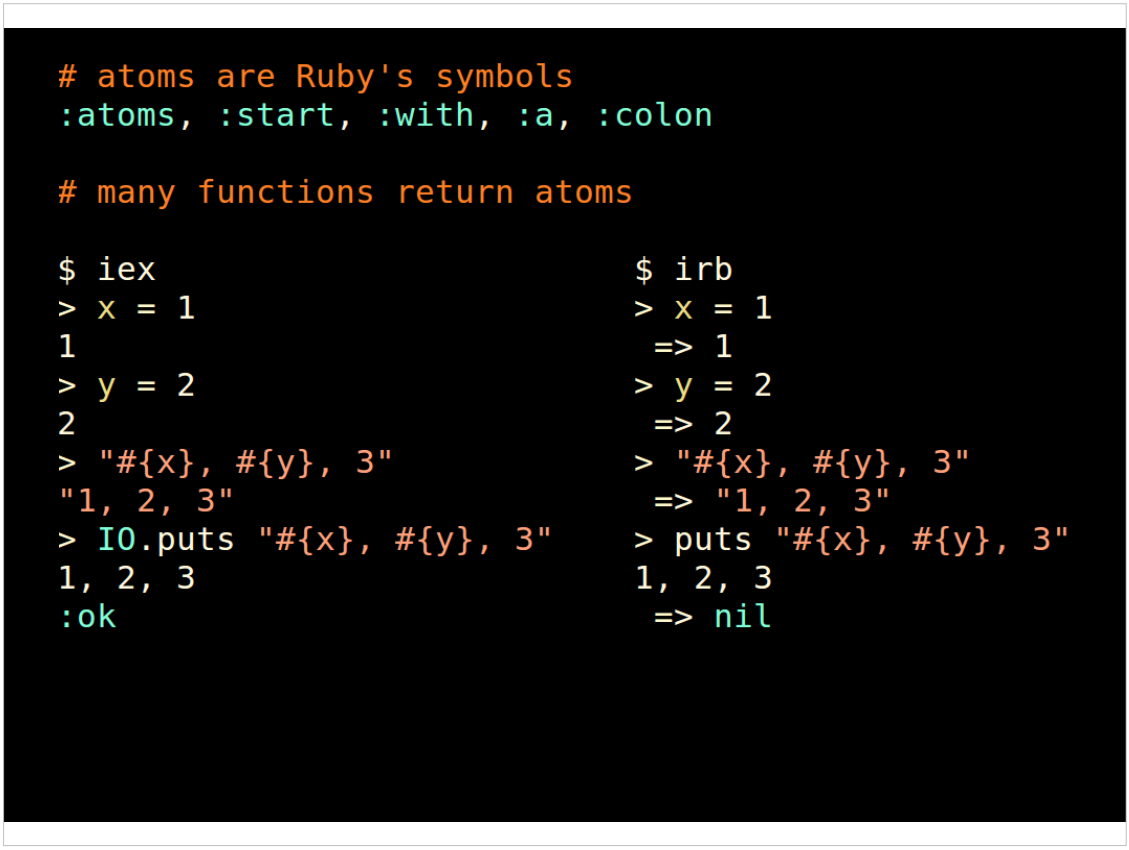
Conditionals are expressions. Their true form is

if (condition, do: (code block))

if condition do
  ...
end

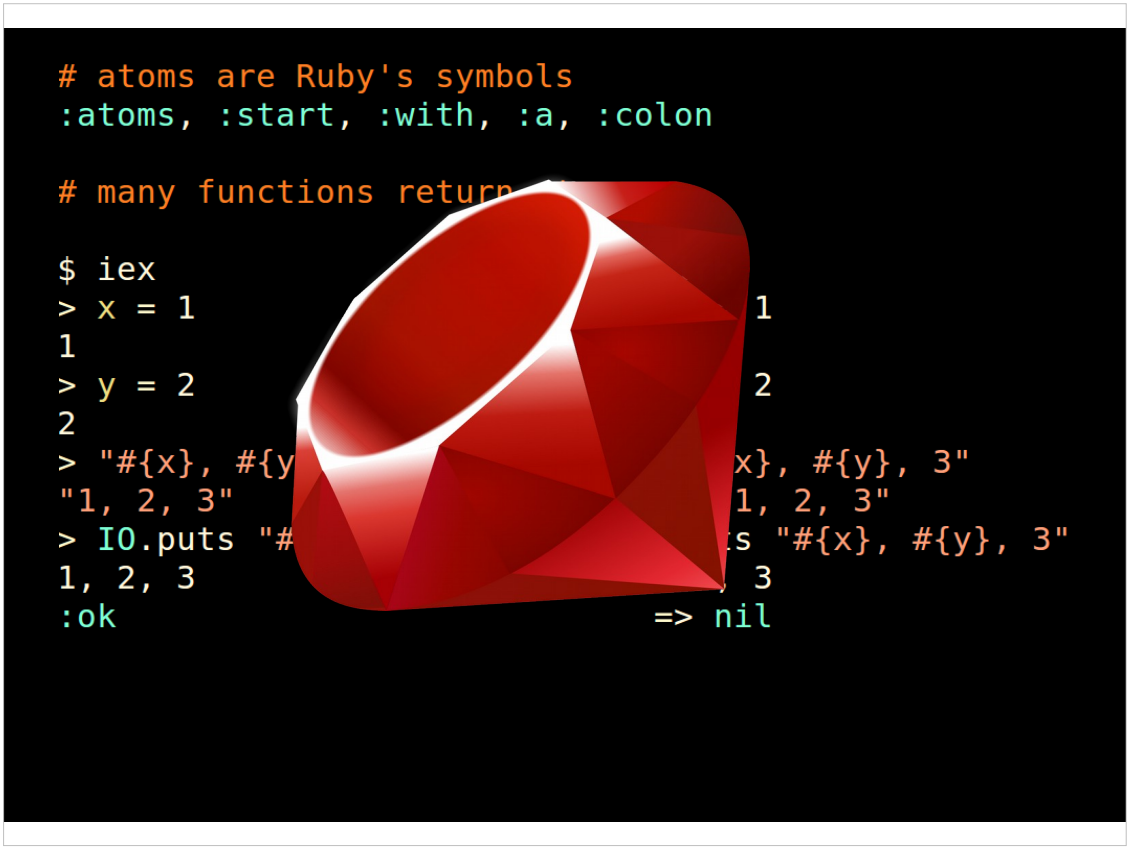
is syntactical sugar.
Actually if and unless are macros (see defmacro).

http://elixir-lang.org/getting_started/5.html



Elixir's atoms and Ruby's symbols are basically the same thing and have the same syntax.

String interpolation is the same.

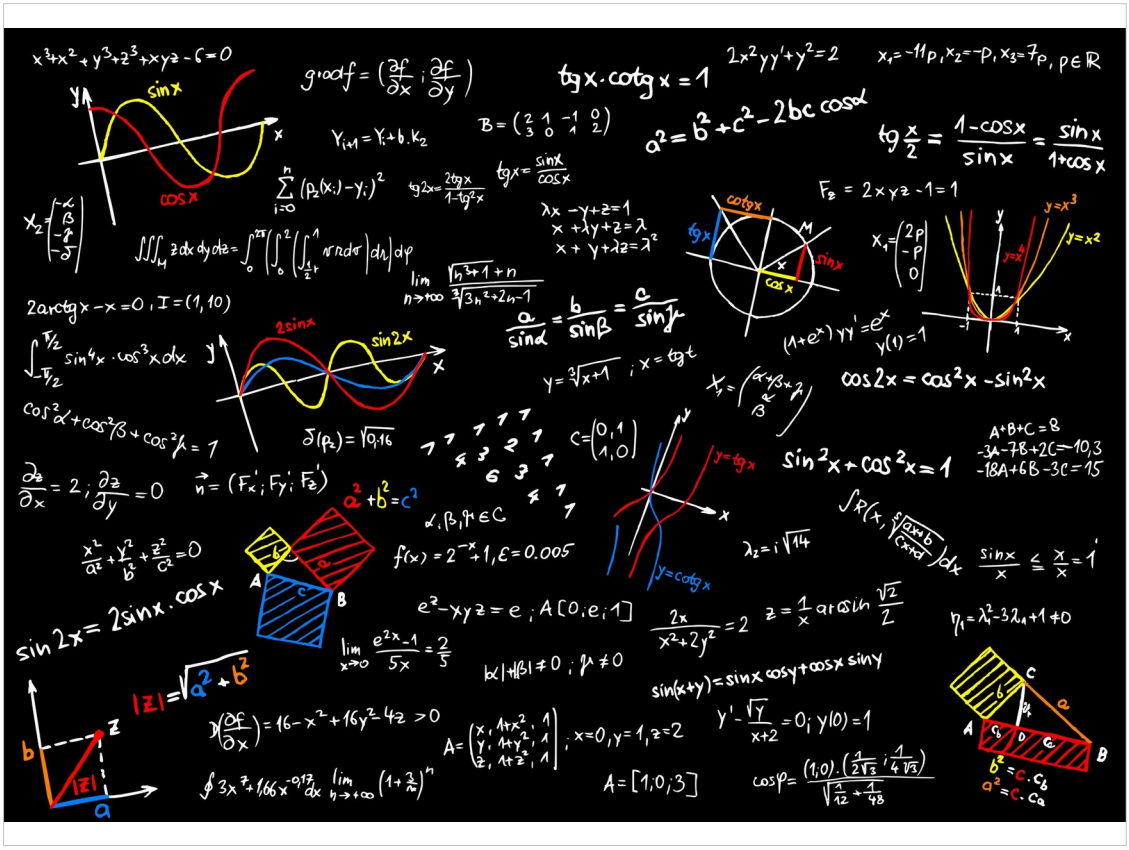


Still quite Ruby-like up to now.

http://commons.wikimedia.org/wiki/File:Ruby_logo.png
Yukihiro Matsumoto, Creative Commons Attribution-Share Alike 2.5 Generic

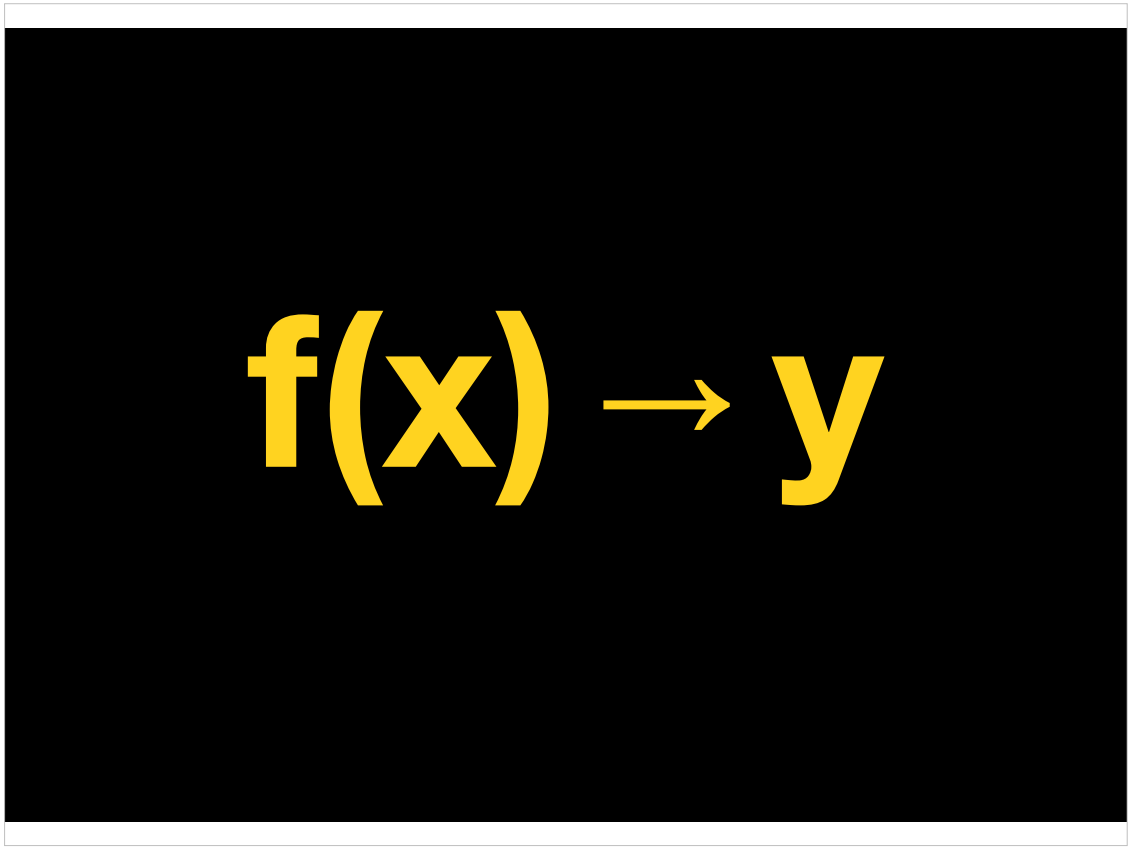
So what are the big differences with Ruby?

http://commons.wikimedia.org/wiki/File:Apples,_Pears,_Oranges.jpg
Mark and Allegra Jaroski-Biava
Creative Commons Attribution-Share Alike 2.0 Generic (cropped)



Elixir is functional. Biggest difference!

http://wall.sf.co.ua/id91895



A function takes one or more arguments and returns one or more values.

It never changes its arguments.

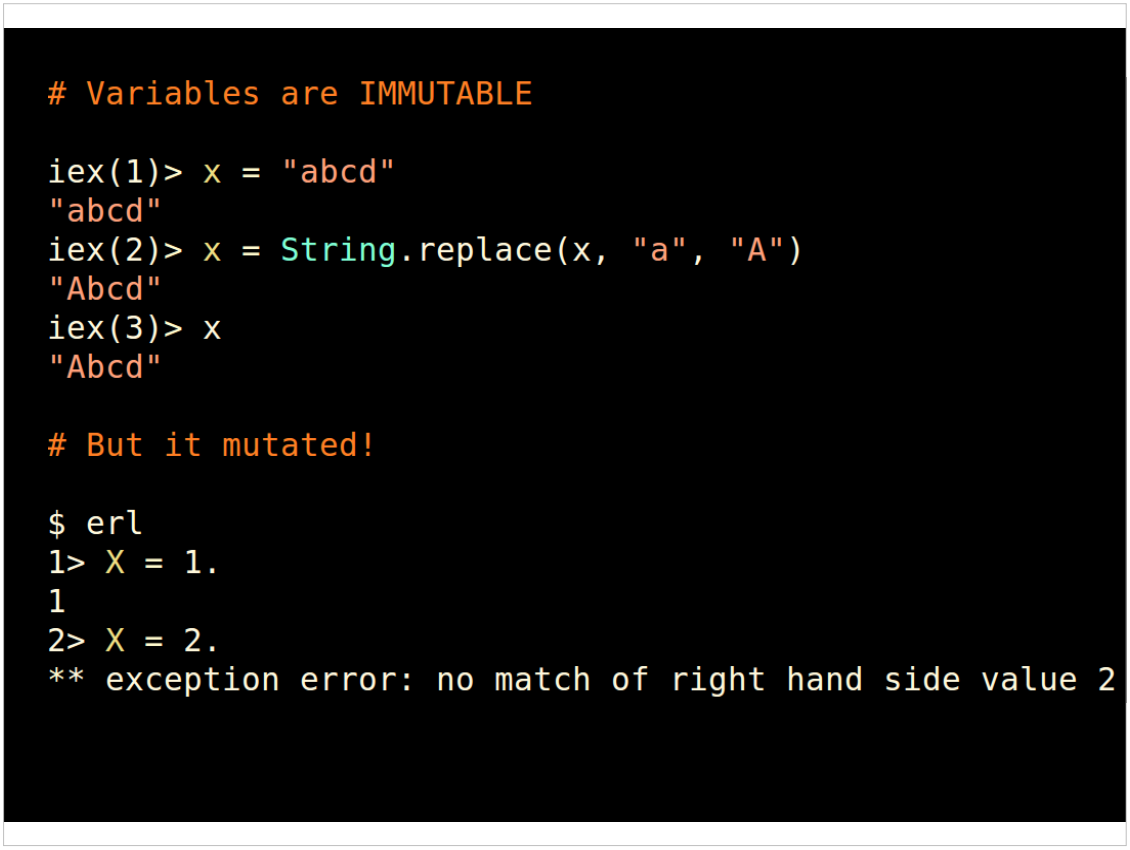
It only returns new values.

OO languages have methods that change the state of the object they are called upon.
Functional languages don't do that.

So the other big difference is: change.

http://pixabay.com/en/change-money-coin-coins-20272/

Actually, the lack of change. Unmutability.



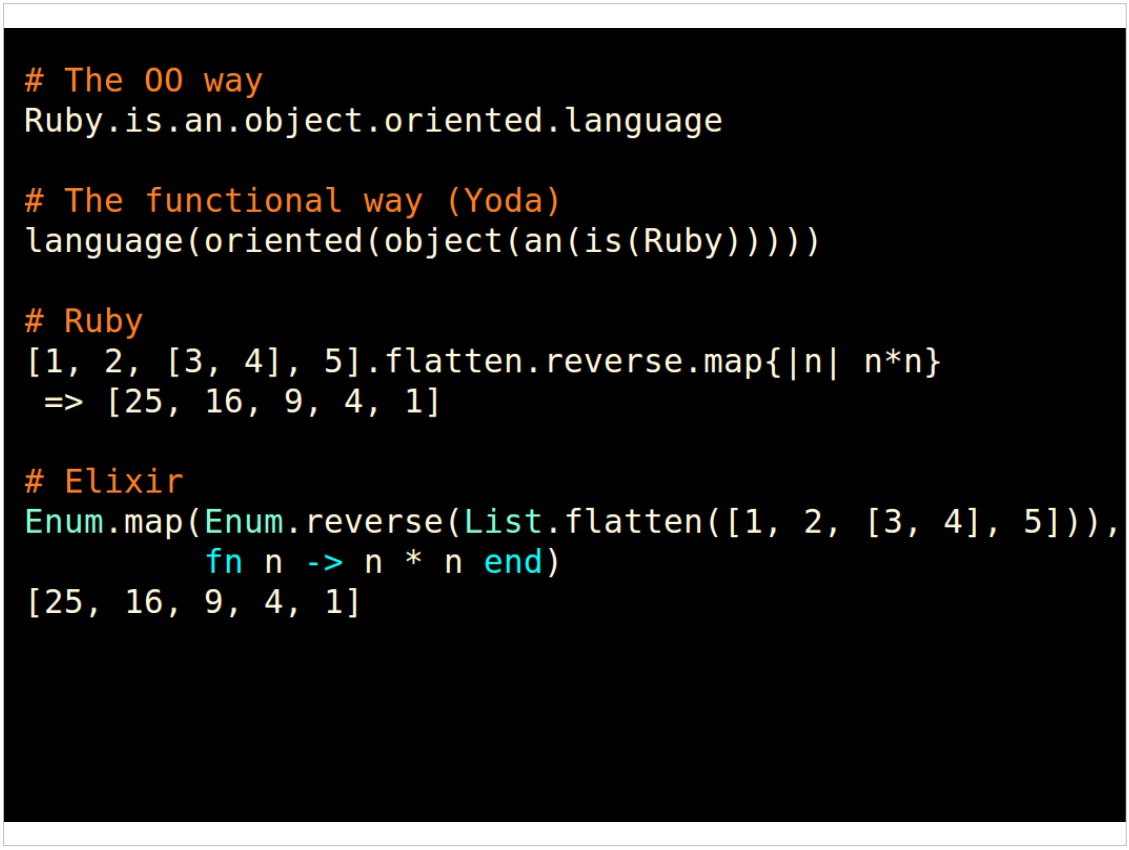
```
# Variables are IMMUTABLE

iex(1)> x = "abcd"
"abcd"
iex(2)> x = String.replace(x, "a", "A")
"Abcd"
iex(3)> x
"Abcd"

# But it mutated!

$ erl
1> X = 1.
1
2> X = 2.
** exception error: no match of right hand side value 2
```

Variables are immutable.
You can't change the value of a variable once you assign a value to it.
Well, did I mutate the value of x? Not really.

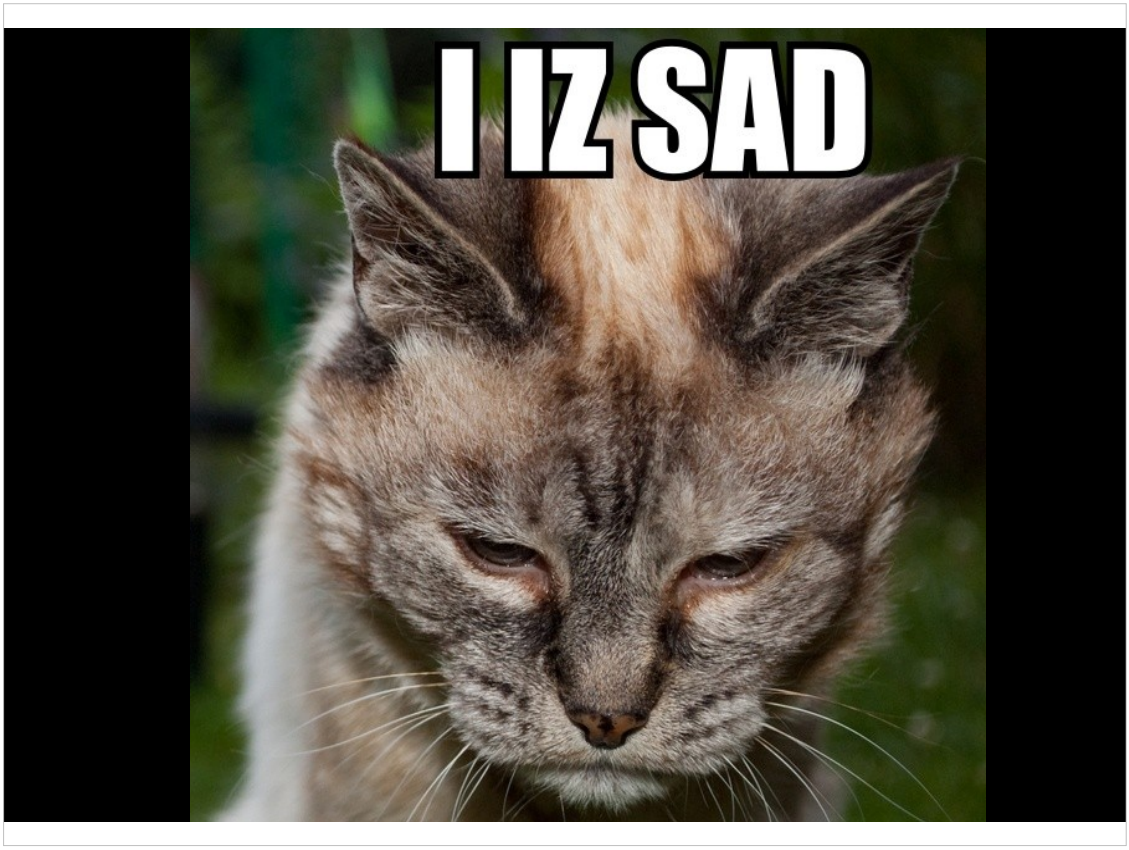
Let's try it in Erlang.
Variables are immutable in Erlang too. Same virtual machine. You try to reassign a variable: error!

What did Elixir do? For our convenience it forgets about the original x variable and lets us use the same name for the variable at lines 2 and 3. But it is a different variable. Mutability is an illusion.

Adding a ^ (the pin operator) as in ^x makes the variable completely immutable.

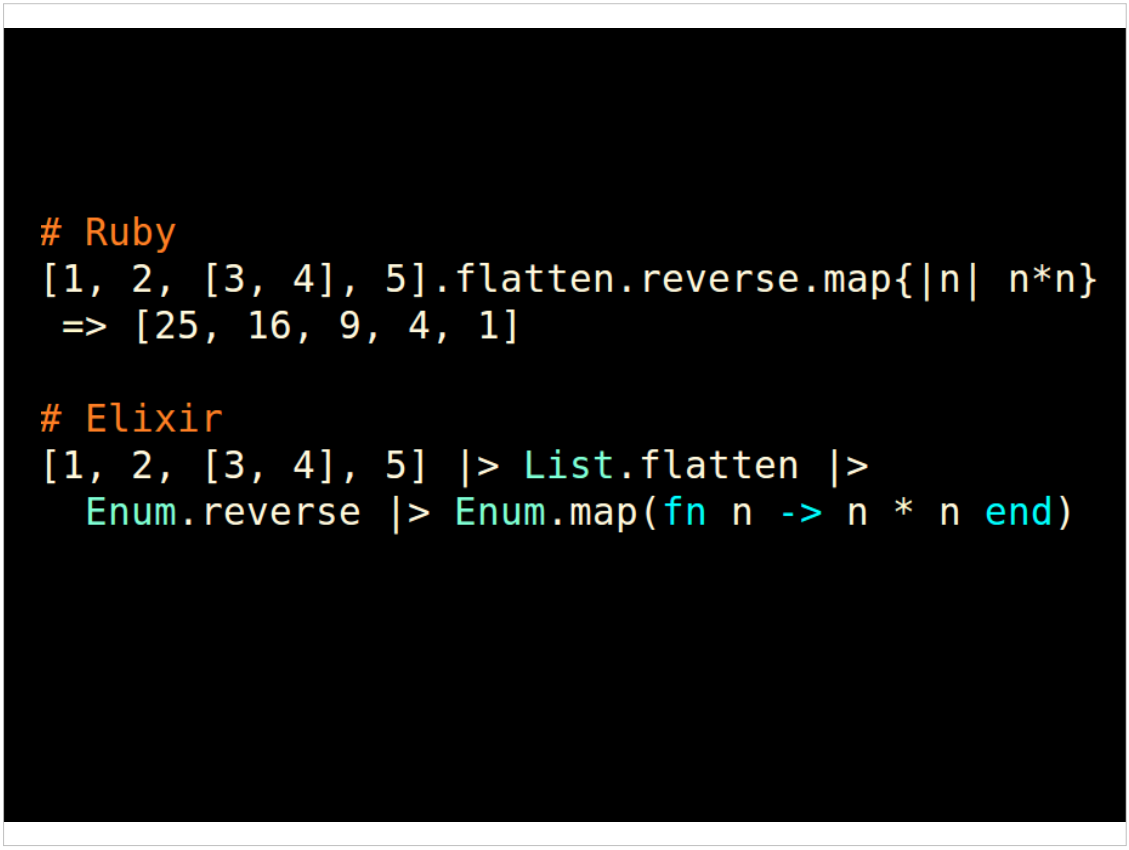


Being functional is very different from OO.
Think in reverse.
It's function2(function1(value), instead of value.method1.method2
Very nice nested function calls, or not so nice?



Tough luck but fortunately this is not the way to do it

http://commons.wikimedia.org/wiki/File:SadCat1.jpg
Dimitri Torterat (Diti) for original photo
Túrelio for derivative
Creative Commons Attribution-Share Alike 3.0 Unported (cropped)



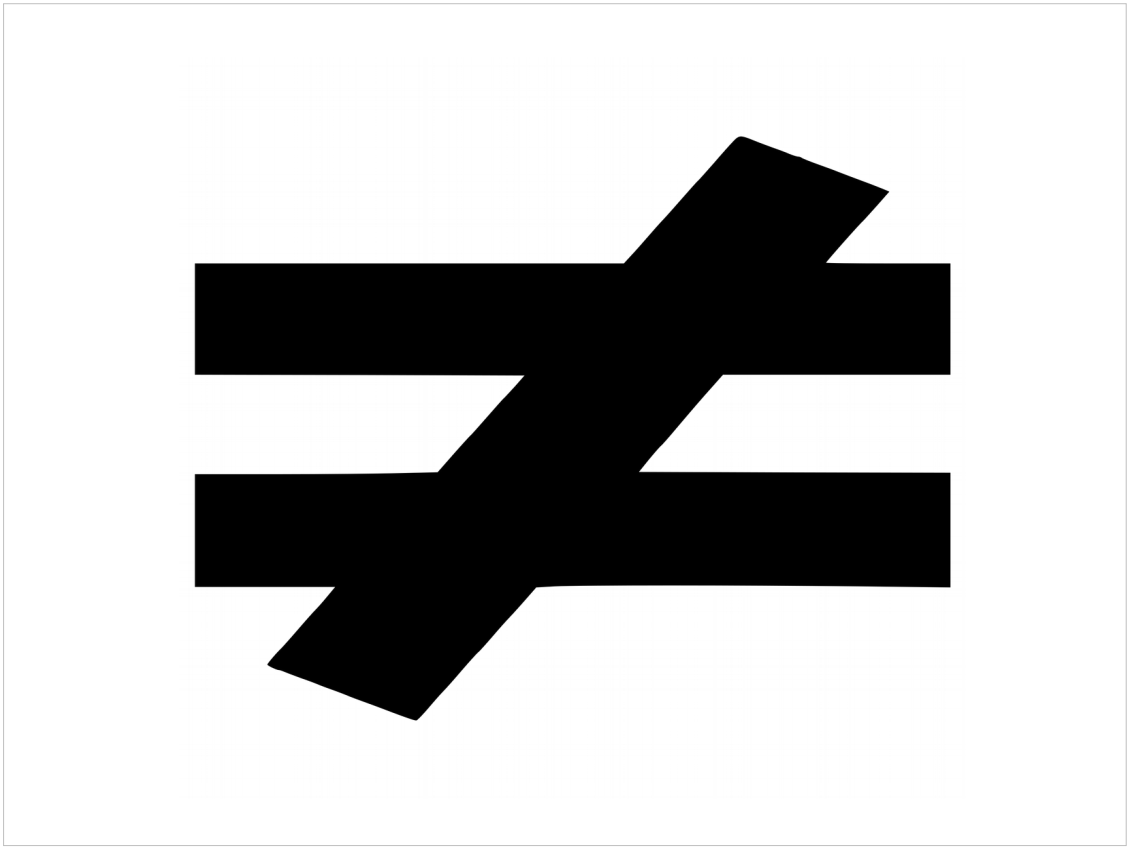
We use the pipe operator to compose functions in a natural way.

```elixir
# Elixir has no loops
defmodule AreYou do
  def bored? do
    Float.floor(:random.uniform * 5, 0) == 0.0
  end
end

defmodule Listen do
  def to_me do
    IO.puts "Listen to me"
    # Recursion with tail optimization
    unless AreYou.bored?, do: Listen.to_me
  end
end

:random.seed(:os.timestamp)
Listen.to_me
```

There is no need for loops.
A loop is made by calling the same function recursively until we reach a return condition.
How about stack overflows? The language automatically performs tail call optimization
http://stackoverflow.com/questions/310974/what-is-tail-call-optimization
No state to store == no frames on the stack.

http://elixir-lang.org/getting_started/9.html

Functions have somewhat similar conventions to Ruby's methods.
Functions that end with ? return booleans.
Functions that end with a bang! raise an exception when they fail.



Third difference.
The assignment is not an assignment.
It's a match operator.

https://openclipart.org/detail/167818/not-equal-to-4-by-dripsandcastle

It means "Do the values on the left side match the values on the right?" If positive, the interpreter assigns unbound variables on the left to the values they match on the right.
That's why x = 3 behaves in a natural way.
x = x + 2 won't work in Erlang.

In 1 c can match :cont
In 2 :stop can't match c, because c has value :cont and :stop is an atom so it's bound by definition.

When c becomes :stop the match in 3 succeeds.
This is a common way to check for return values from functions and this is why many functions return atoms.
http://elixir-lang.org/getting_started/4.html

By the way, {we, introduced, tuples}

```
# read a file
case File.read "/home/me/elixir/doc.txt" do
  {:ok, content}  -> do_something_with(content)
  {:error, error} -> log(error)
end

# get a web page
:inets.start()
{:ok, {status, headers, content}} =
  :httpc.request "http://www.example.com"
{:ok, file} = File.open "index.html", [:write, :utf8]
IO.binwrite file, content
File.close file
```

Reading a file has the same syntax as in Ruby.
The case statement works with matches.
Check the cond statement too.

:library.function is a way of calling an Erlang library, much like what we can do in JRuby with Java libraries.

:httpc.requests and File.open return tuples.
If those tuples don't match the ones on the left the program halts immediately.

Pattern matching is the reason why exceptions are rarely used in Elixir but check try catch rescue throw raise after at http://elixir-lang.org/getting_started/17.html

Fourth difference: strings.

http://commons.wikimedia.org/wiki/File:Piano_strings_6.jpg
Alan Levine
Creative Commons Attribution 2.0 Generic
(cropped)

```
iex(1)> "a" == "a"
true
iex(2)> 'a' == 'a'
true
iex(3)> "a" == 'a'
false
iex(4)> "è utf8" # a string
"è utf8"
iex(5)> 'è utf8' # list of characters
[232, 32, 117, 116, 102, 56] # [ ] are lists
iex(6)> to_char_list "è utf8"
[232, 32, 117, 116, 102, 56]
iex(7)> to_string 'è utf8'
"è utf8"
iex(8)> IO.puts '#{x}, #{y}' # '' interpolate
1, 2
:ok
iex(9)> "foo" <> "bar"       # ugly as hell
"foobar"
```

Double quotes and single quotes are different.

Lists of characters can be built from strings and strings can be built from lists of characters.

Square brakets define lists.

Interpolation works even in single quotes.

Concatenation has a ugly syntax: <> which is the concatenation operator for binary data

<<0, 1>> <> <<2, 3>>

Ugh!

http://elixir-lang.org/getting_started/6.html

http://en.wikipedia.org/wiki/File:Beer_Cans-1.jpg
Visitor7
Creative Commons Attribution-Share Alike 3.0 Unported (cropped, darkened)

```
# This is a list not an array
iex(1)> x = [1, 2, 3]
[1, 2, 3]

# Concatenation
iex(2)> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]

# Head and tail of a list
iex(3)> [y|z] = x
[1, 2, 3]
iex(4)> y   # head
1
iex(5)> z   # tail
[2, 3]
iex(6)> List.last(x)
3
```

[ ] enclose lists.

Concatenation operators are not overloaded. They can be quite ugly. Remember the one for strings.

Lists have a head and a tail.
Very Lispy

So many more things to know about Elixir

http://www.flickr.com/people/jamescridland/
James Cridland http://james.cridland.net/
CC BY 2.0 (cropped)

```elixir
keyword_list = [{:a, "a"}, {:b, "b"}]
map = %{:a => "a", 2 => "b"}

@moduledoc
@doc

defmodule Language do
  defstruct name: "Ruby", age: 19
end

# comprehension, not a loop
for dir  <- dirs,
    file <- File.ls!(dir),
    path = Path.join(dir, file),
    File.regular?(path) do
  File.rm!(path)
end
```

Keyword lists
Maps (Ruby's hashes)
http://elixir-lang.org/getting_started/7.html

Annotations to insert tests and documentation into the code.
http://elixir-lang.org/getting_started/14.html
http://elixir-lang.org/getting_started/mix_otp/9.html

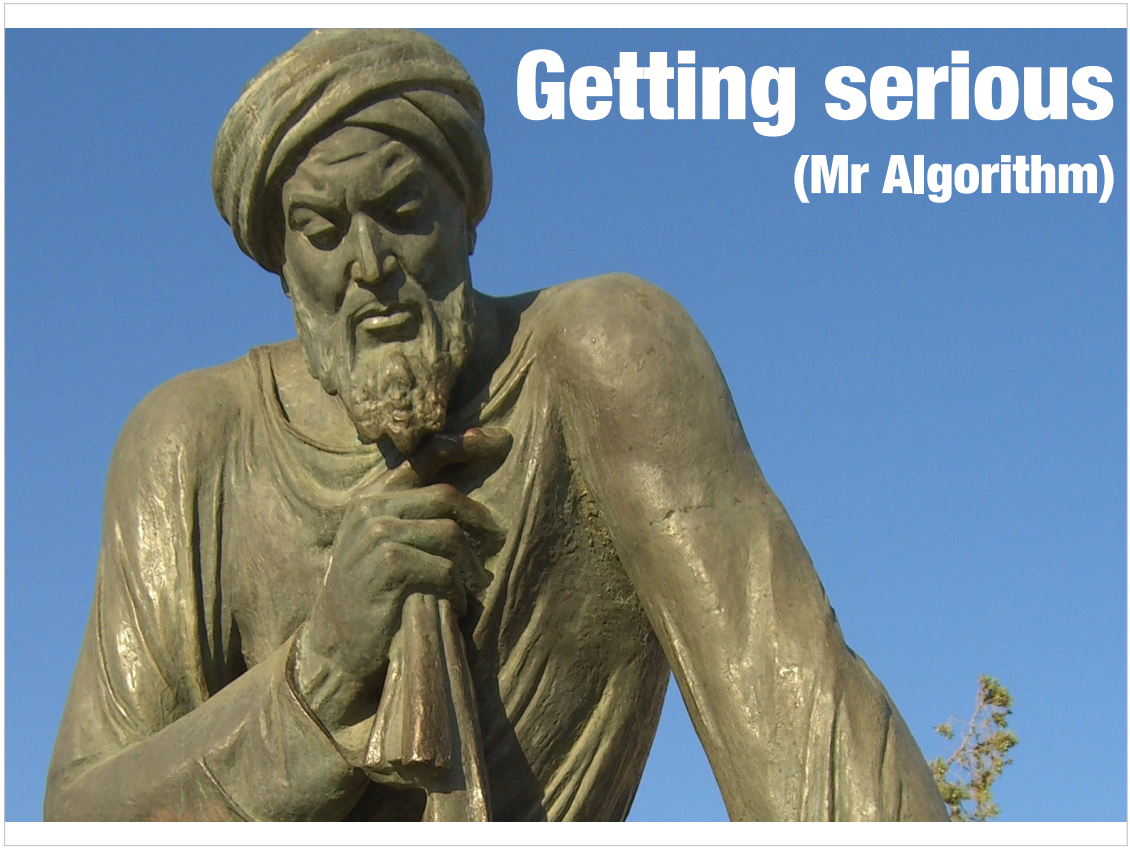
Modules.
http://elixir-lang.org/getting_started/8.html

Structs.
http://elixir-lang.org/getting_started/15.html

Comprehension.
http://elixir-lang.org/getting_started/18.html

But time is running out

http://commons.wikimedia.org/wiki/File:Time_is_running_out.jpg
Sergey Galyonkin



Statue of Muḥammad ibn Mūsā al-Khwārizmī at Khiva, Uzbekistan.
One of the fathers of algebra. His book “On the Calculation with Hindu Numerals” was translated into Latin as “Algoritmi de numero Indorum” mixing the transliterated author name with the title. By confusion we are talking about algorithms now.

http://en.wikipedia.org/wiki/Mu%E1%B8%A5ammad_ibn_M%C5%ABs%C4%81_al-Khw%C4%81rizm%C4%AB

```
#!/usr/bin/env elixir
defmodule Server do
  def echo do
    # actor based concurrency
    receive do
      {client, message} -> IO.puts "server: #{message}"
         send client, message
    end
    echo # Tail recursion http://xkcd.com/1270/
  end
end

# could spawn to a different machine by Node.connect
pid = spawn fn -> Server.echo end
send pid, {self, "Hi"}
receive do
  message -> IO.puts "client: #{message}"
end

$ ./spawn.exs
server: Hi
client: Hi
```

A client/server.
The server receives tuples {client, message} and puts them to stdout.
Then it loops on itself.
Tail recursion to the rescue!
The client spawns the server process, sends its own id and a message, it waits for the response.
http://elixir-lang.org/getting_started/11.html

To compile:
$ elixirc spawn.exs
It generates Elixir.Server.beam
Then you can use the Server module in iex
$ iex
> pid = spawn fn -> Server.echo end
> send pid, {self, “Hi”}
> ...

```
$ git clone https://github.com/phoenixframework/phoenix.git
$ cd phoenix

# mix is rake + bundle
# mix.exs is Rakefile + Gemfile
# mix.lock is Gemfile.lock
$ mix do deps.get, compile

# From this directory! Important!
$ mix phoenix.new my_project ~/my_project

$ cd ~/my_project
$ mix do deps.get, compile  # bundle
$ mix phoenix.start         # rails s

http://localhost:4000

$ mix help                  # rake -T
$ mix phoenix.routes        # rake routes
```

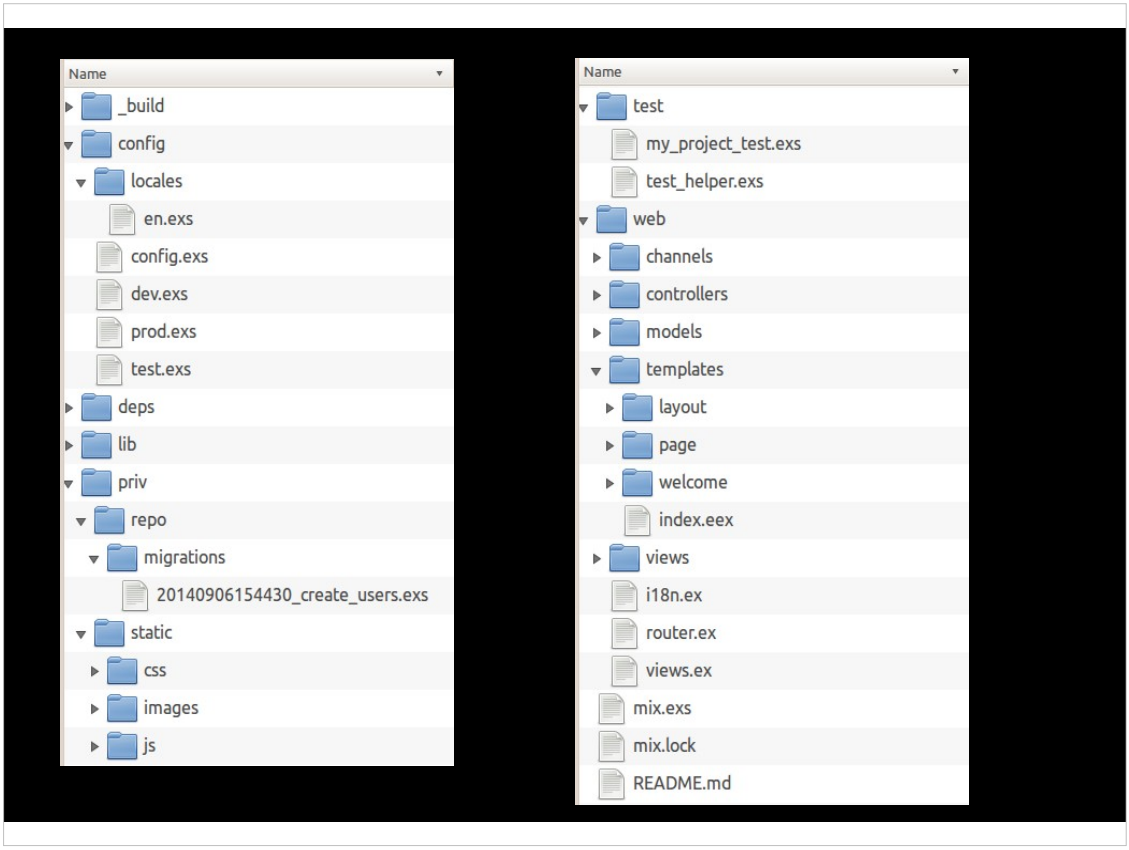
$ git clone
https://github.com/phoenixframework/phoenix.git
$ cd phoenix
$ mix do deps.get, compile
# From this directory! Important!
$ mix phoenix.new my_project ~/my_project
$ cd ~/my_project
$ mix do deps.get, compile
$ mix phoenix.start
http://localhost:4000
$ mix help # rake -T
$ mix phoenix.routes # rake routes
# mix deps.clean –all is nice to know
Other web frameworks
https://github.com/elixir-web/weber # Rails like
https://github.com/dynamo/dynamo # Sinatra like

```
defmodule MyProject.Mixfile do
  use Mix.Project

  def project do
    [ app: :my_project,
      version: "0.0.1",
      elixir: "~> 1.0.0-rc1",
      elixirc_paths: ["lib", "web"],
      deps: deps ]
  end

  def application do
    [
      mod: { MyProject, [] },
      applications: [:phoenix, :cowboy, :logger]
    ]
  end

  defp deps do
    [
      {:phoenix, "0.4.0"},
      {:cowboy, "~> 1.0.0"}
    ]
  end
end
```

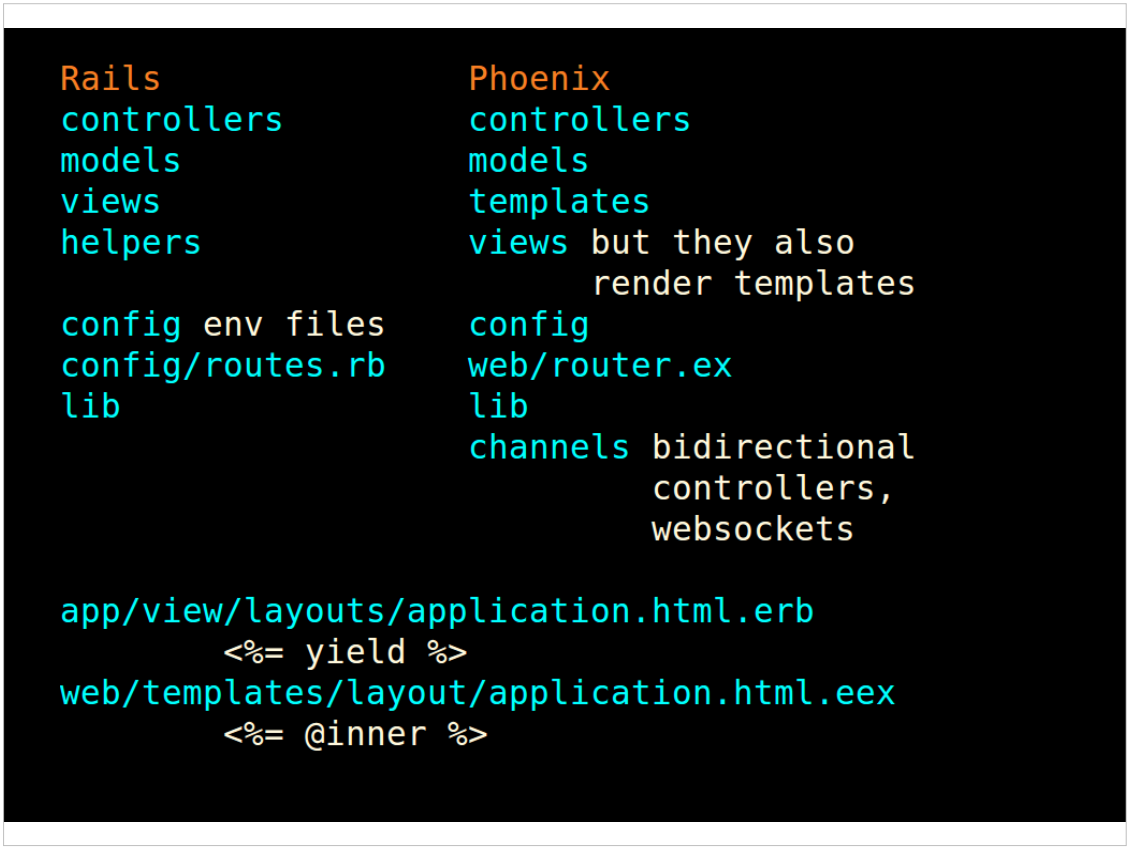
```
%{"cowboy": {:package, "1.0.0"},
  "cowlib": {:package, "1.0.0"},
  "linguist": {:package, "0.1.2"},
  "phoenix": {:package, "0.4.0"},
  "plug": {:package, "0.7.0"},
  "poison": {:package, "1.0.3"},
  "ranch": {:package, "1.0.0"}}
```

$ git clone https://github.com/phoenixframework/phoenix.git
$ cd phoenix
$ mix do deps.get, compile
# From this directory! Important!
$ mix phoenix.new my_project ~/my_project
$ cd ~/my_project
$ mix do deps.get, compile
$ mix phoenix.start

http://localhost:4000



The structure of a phoenix application.



https://github.com/phoenixframework/phoenix#channels

Young framework.
Many tools still lacking.
Lots of DIY.

SElephant
http://zh.wikipedia.org/wiki/File:Basic_DIY_Tools.jpg

```
$ psql -U postgres
# create role my_project login password 'password';
CREATE ROLE
# create database my_project owner my_project
encoding='UTF8' lc_collate='en_US.UTF-8' lc_ctype='en_US.UTF-8';
CREATE DATABASE
# grant all on database my_project to my_project;
GRANT
# \q

$ vi mix.exs
defp deps do
  ...
  {:postgrex, ">= 0.5.0"}
  ...
end
```

Some rough edges.
The database must be added manually.

```
$ vi mix.exs
defp deps do
  ...
  {:ecto, "~> 0.2.0"}
  ...
end

$ vi lib/my_project/repo.ex
  defmodule Repo do
    use Ecto.Repo, adapter: Ecto.Adapters.Postgres

    def conf do
      parse_url "ecto://my_project:postgres@localhost/my_project"
    end

    def priv do
      app_dir(:my_project, "priv/repo")
    end
  end

$ mix ecto.gen.migration Repo create_users
Compiled lib/my_project/repo.ex
Generated my_project.app
* creating priv/repo/migrations
* creating priv/repo/migrations/20140906154430_create_users.exs
```

Migrations and the DB adapter must be added.
Use the ecto database wrapper.

https://github.com/elixir-lang/ecto
http://elixir-lang.org/docs/ecto/

We have:
belongs_to
has_many
DSL for select / insert / update / transactions but not for creating / dropping tables

```
# No DSL yet!
# The up and down functions must return the SQL to execute

$ vi priv/repo/migrations/20140906154430_create_users.exs
defmodule Repo.Migrations.CreateUsers do
  use Ecto.Migration

  def up do
    "CREATE TABLE users(id serial primary key, content varchar(140))"
  end

  def down do
    "DROP TABLE users"
  end
end

$ mix ecto.migrate Repo
* running UP _build/dev/lib/my_project/priv/repo/migrations/201409061
54430_create_users.exs
```

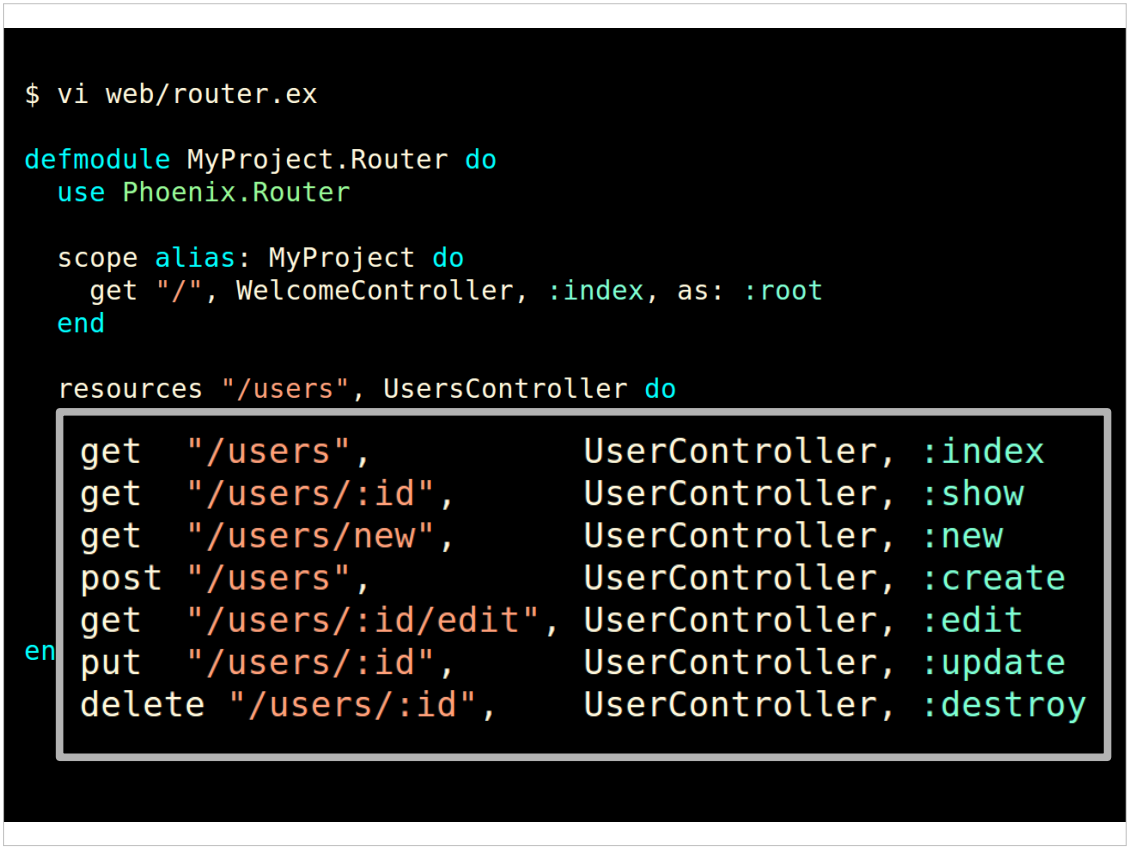
We don't have yet:

DSL in migrations

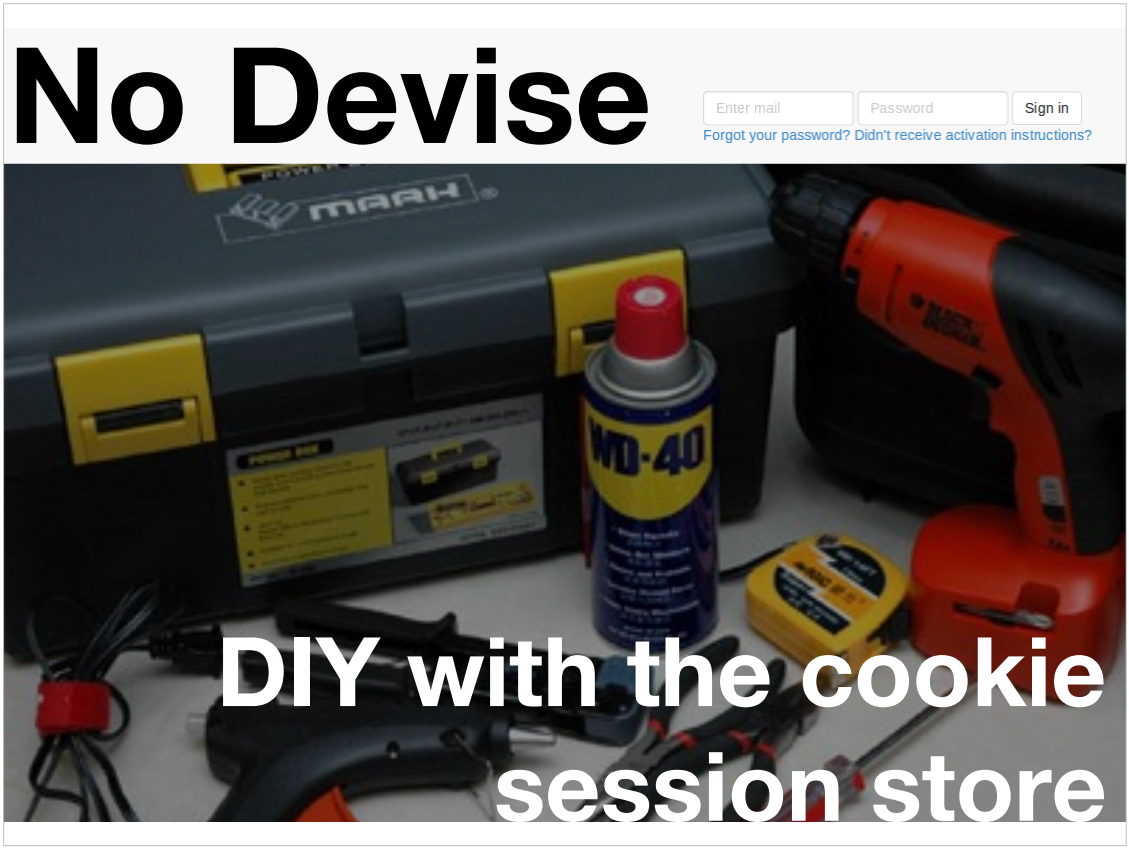
```
$ vi web/router.ex

defmodule MyProject.Router do
  use Phoenix.Router

  scope alias: MyProject do
    get "/", WelcomeController, :index, as: :root
  end

  resources "/users", UsersController do
    resources "/pages", PagesController
  end

  scope path: "/admin", alias: MyProject.Admin, helper: "admin" do
    resources "/users", UsersController
  end

end
```

Routes are restful and can be nested

We have scopes and static routes.



The usual restful ~~methods~~ functions in controllers



No authentication frameworks yet but phoenix has a cookie based session store that can be used to store the user id https://github.com/elixir-lang/plug/blob/master/lib/plug/session.ex

See my implementation with a plug in the github demo app.

SElephant
http://zh.wikipedia.org/wiki/File:Basic_DIY_Tools.jpg
Attribution-ShareAlike 3.0 Unported (cropped, darkened)

```elixir
defmodule MyProject.User do
  use Ecto.Model
  import Ecto.Query

  schema "users" do
    field :email,    :string
    field :password, :string
  end

  validate user,
    email:    present(),
    password: present()

  def encrypt_password(plaintext) do
    :base64.encode(:crypto.hash(:sha256, to_char_list(plaintext)))
  end

  def find(email, plaintext_password) do
    encrypted_password = encrypt_password(plaintext_password)
    query = from u in MyProject.User,
      where: u.email == ^email and u.password == ^encrypted_password,
      select: u
    Repo.all(query)
  end
end
```

There is no ActiveRecord magic to define attributes from the database schema.

There are validations (with ecto).

A look to the functions:

1) Strings passed to Erlang must be converted to char lists.

2) The ^ pin operator is important inside Ecto queries. Anything without a pin is a variable of the query, not variables of the function.
It reminds of how to inject local variables inside a squeel block in Ruby.

```elixir
defmodule MyProject.Admin.UsersController do
  use Phoenix.Controller
  require Logger
  alias MyProject.User
  require Authentication
  require AdminsOnly

  plug Authentication
  plug AdminsOnly

  def show(conn, _params) do
    %{ "id" => user_id } = _params
    { user_id, _ } = Integer.parse(user_id)
    user = Repo.get(User, user_id)
    render conn, "show", user: user
  end
end
```

The example has no error checks on params and on the result of Repo.get

_params with the underscore is the idiomatic way, sorry for that.

Authentication and AdminsOnly are two plugs (kind of before actions) that implement authentication and authorization. Custom built for this application. Check the code on github.

```
<ul>
<%= if @current_user == nil do %>
  <li><a href="<%= Router.sessions_path(:new) %>">Login</a></li>
<% else %>
  <li><a href="<%= Router.users_path(:show, @current_user.id) %>">
      <%= @current_user.email %></a></li>
  <li><a href="<%= Router.sessions_path(:destroy) %>">Logout</a></li>
<% end %>
</ul>

<%= for notice <- Flash.get_all(@conn, :notice) do %>
  <div class="container">
    <div class="row">
      <div class="alert alert-success" role="alert">
        <p><%= notice %></p>
      </div>
    </div>
  </div>
<% end %>
```

The equal in <%= if cond do %> is important

The default .eex which is similar to .erb
Haml and Slim are available too.
https://github.com/phoenixframework/phoenix#template-engine-configuration

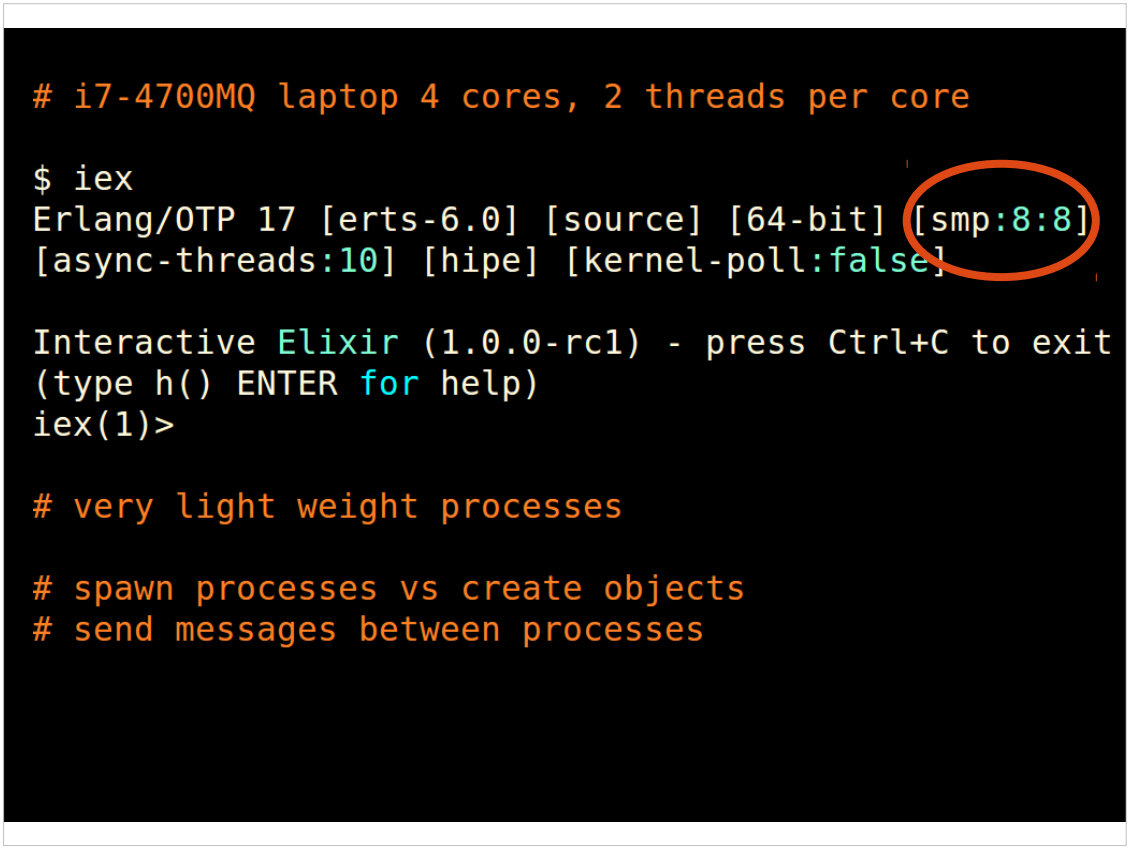
if blocks return a string. Without the equal in <%= the if will execute but its return value (the HTML) won't be rendered.

The for loop is actually a comprehension.
Flash is set in controllers like this

```
def destroy(conn, _params) do
  fetch_session(conn)
  |> delete_session(:user_id) # this is to logout
  |> Flash.put(:notice, "Logout successful")
  |> redirect Router.pages_path(:index)
end
```

Is this Elixir better than Ruby?

http://en.wikipedia.org/wiki/File:Benjamin_G_Bowden_-_Spacelander_Bicycle.jpg
Brooklyn Museum
CC BY 3.0 (cropped)



Hard to tell but Elixir has a definite advantage when parallelism is important.

It accesses all the cores of the CPU.

It has very light weight processes. You can have thousands of them running inside your application.

Consider changing your approach: create processes to store state and make them exchange messages.

Thanks to a Ruby like syntax is an easy entry point into the world of functional languages.

10 REM SINCLAIR ZX81 KEYS
20 PRINT “MY FIRST COMPUTER”
30 PRINT “NOT TOO FUNCTIONAL BUT FUNNY”

en.wikipedia.org/wiki/File:Sinclair-ZX81.png
Evan-Amos

Download this presentation from
http://connettiva.eu/rubyday
Plus links to elixir and phoenix resources
Plus three HOWTOs to install erlang, elixir, phoenix.

Phoenix demo app at
https://github.com/pmontrasio/phoenix-demo-app