Hello Pyladies!

Elixir for Pythonistas

About me

Hey, I am Martin Wiso :-)

- Born in Prague Czech Republic
- Living in Berlin since 2012
- Working mostly as Backend Senior Software Engineer
- Using multiple languages and trying new one (almost) every year
- Specializing in building and scaling cloud-native services
- Experience in building mission-critical systems
- Building and leading small engineering teams

Agenda

- Overview of Functional Programming, Elixir and friends
- Language basics
- Advanced topics
- Break
- Hands on session

There is no such thing as the language

- Python is great for learning programming and building small to large programs

There is no such thing as the language

- Python is great for learning programming and building small to large programs
- Elixir/Erlang is great language for building complex (distributed) systems

There is no such thing as the language

- Python is great for learning programming and building small to large programs
- Elixir/Erlang is great language for building complex (distributed) systems
- Great value in learning any new language

There is no such thing as the language

- Python is great for learning programming and building small to large programs
- Elixir/Erlang is great language for building complex (distributed) systems
- Great value in learning any new language
- Especially language that will force you think and design programs differently

There is no such thing as the language

- Python is great for learning programming and building small to large programs
- Elixir/Erlang is great language for building complex (distributed) systems
- Great value in learning any new language
- Especially language that will force you think and design programs differently
- Learnings should influence way you program in general

There is no such thing as the language

- Python is great for learning programming and building small to large programs
- Elixir/Erlang is great language for building complex (distributed) systems
- Great value in learning any new language
- Especially language that will force you think and design programs differently
- Learnings should influence way you program in general
- Enough said let's start with journey into functional programming and Elixir….

What is Functional Programming?

Functional programming is one of many different approaches to express our thoughts to computers.

What is Functional Programming?

Functional programming is one of many different approaches to express our thoughts to computers.

It is quite old concept from mathematics - Lambda calculus and Category theory

CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

Function purity can be taken too far. If no state is modified, then a program might as well not have been run at all. (Source: xkcd.com)

What is Functional Programming?

Functional programming is one of many different approaches to express our thoughts to computers.

It is quite old concept from mathematics - Lambda calculus and Category theory

One the first language that was functionally flavoured is LISP

What is Functional Programming?

Functional programming is one of many different approaches to express our thoughts to computers.

It is quite old concept from mathematics - Lambda calculus and Category theory

One the first language that was functionally flavoured is LISP

There are many other functional languages eg Haskell, OCaml ... SQL, XSLT

What is Functional Programming?

Functional programming is one of such different approaches to express our thoughts to computers.

It is quite old concept from mathematics - Lambda calculus and Category theory

One the first language that was functionally flavoured is LISP

There are many other functional languages eg Haskell, OCaml ... SQL, XSLT

Also imperative languages are incorporating some of functional features like lambda

Compare simple iterative and functional example

Python version:

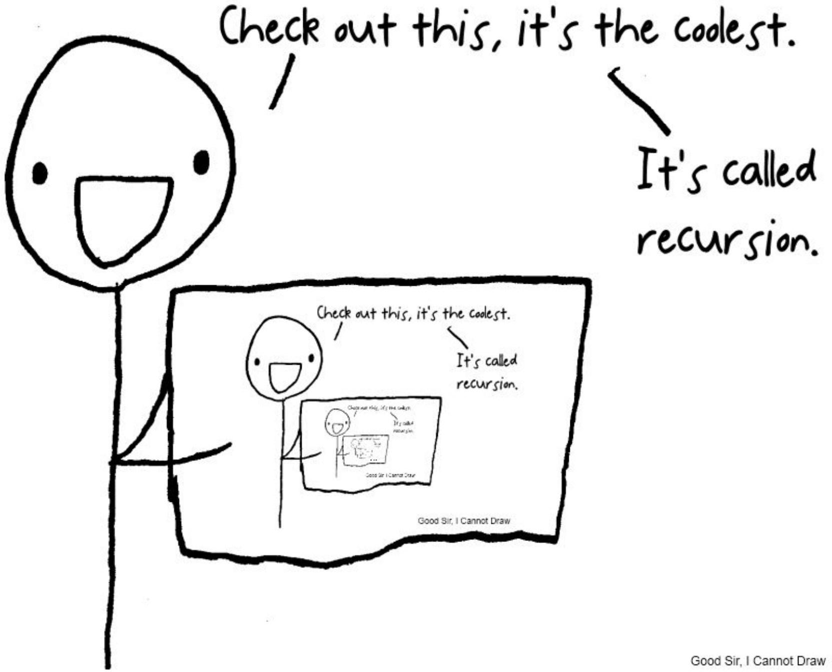
```
def double_each(a_list):
   for index, a in enumerate(a_list):
      a_list[index] = a * 2

   return a_list
…
In[1]:  double_each([1, 2, 3])
Out[2]: [2, 4, 6]
```

Elixir version:

```
defmodule Math do
  def double_each([head | tail]) do
    [head * 2 | double_each(tail)]
  end
  def double_each([]), do: []
end
…
iex(1)> Math.double_each([1, 2, 3])
[2, 4, 6]
```

Recursion all the things!



Source (https://github.com/kblake/functional-programming/blob/master/README.md)

What is Elixir?

Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

What is Elixir?

Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

Elixir leverages the BEAM virtual machine, known for running low-latency, distributed and fault-tolerant systems build by Ericsson primarily for telecommunication industry.

What is Elixir?

Elixir is a dynamic, functional language designed for building scalable and maintainable applications.

Elixir leverages the BEAM virtual machine, known for running low-latency, distributed and fault-tolerant systems build by Ericsson primarily for telecommunication industry.

Josè Valim (the author of Elixir) wanted to bring a comfortable Ruby-like experience on top of BEAM VM as Ruby tends to be difficult to scale for heavy load.

Demo time

Distributed capabilities of BEAM VM

Elixir basics

Language specifics

- Dynamically typed functional language

```
iex(1)> a = 1
1

iex(2)> a = "two"
"two
```

Language specifics

- Dynamically typed functional language
- Immutable data structures

```
iex(1)> m = Map.new()
%{}

iex(2)> m = Map.put(m, :a, 1)
%{a: 1}
```

```
# mutation
my_dict = {}
my_dict['a'] = 1
```

Language specifics

- Dynamically typed functional language
- Immutable data structures
- Pattern matching

```
iex(1)> {a, b, c} = {:hello, "world", 42}
{:hello, "world", 42}
iex(2)> a
:hello
iex(3)> b
"world"
```

Language specifics

- Dynamically typed functional language
- Immutable data structures
- Pattern matching
- Higher-order Functions

```
iex(1)> foo = fn (name) -> IO.puts "Language: #{name}" end
#Function<6.99386804/1 in :erl_eval.expr/5>

iex(2)> iex(12)> print_helper = fn (fun) -> fun.("Elixir") end
#Function<6.99386804/1 in :erl_eval.expr/5>

iex(3)> print_helper.(foo)
Name: Elixir
:ok
```

Language specifics

- Dynamically typed functional language
- Immutable data structures
- Pattern matching
- Higher-order Functions
- No classes just modules, structures and functions

```
iex(14)> defmodule Hello do
...(14)>    def world(), do: IO.puts "Hey!"
...(14)> end
{:module, Hello, <…>, {:world, 0}}
iex(15)> Hello.world()
Hey!
:ok
```

Language specifics

- Dynamically typed functional language
- Immutable data structures
- Pattern matching
- Higher-order Functions
- No classes just modules and functions
- Compose program using functions and pipe operator `|>`

```
"Yes"
|> String.downcase()
…
"yes"
```

Language specifics

- Dynamically typed functional language
- Immutable data structures
- Pattern matching
- Higher-order Functions
- No classes just modules and functions
- Compose program using functions and pipe operator `|>`
- Lightweight processes (not OS threads)

```
iex(1)> pid = spawn fn -> 1 + 2 end
#PID<0.44.0>
```

Language specifics

- Dynamically typed functional language
- Immutable data structures
- Pattern matching
- Higher-order Functions
- No classes just modules and functions
- Compose program using functions and pipe operator `|>`
- Lightweight processes (not OS threads)
- Easy way of sending messages (similar to Actors) in between processes

```
iex(1)> send(pid, "Hello, Joe")
iex(2)> "Hello, Joe"
```

But there is more...

- Interactive shell (REPL **`iEx)`** for quick experimentation
- Way to connect to remote machine running VM into that shell
- This enables you to inspect running system (also change!)

But there is even more...

- Runtime contains tools and libraries for building resilient systems part of runtime for example HTTP client, SFTP client/server, crypto and more...
- Runtime introspection tools that helps observing your running program (see **`observer`**)

But there are even common things ...

- Package manager
- Code formatters, linters, static code analysis tools (external libraries)
- IDE and editor support
- **Most importantly very friendly and helpful community!**

Basic data structures

- Strings/Binaries
- Integers
- Floats
- Atoms
- Boolean
- Pid
- Function

- Lists and Enumerables
- Tuples
- Keywords
- Map and MapSet
- Streams
- Structs
- ...

Strings

```
iex> "hellö"
"hellö"

# String interpolation
iex(1)> world = "World"
iex(2)> "Hello, #{world}"
"Hello, World"

# String concatenation operator
iex(2)> "Hello, " <> " Elixir!"
"Hello, Elixir!"

iex(3)> String.upcase("hellö")
"HELLÖ"
```

```
# Yes it string is binary
iex(5)> is_binary("hellö")
true
```

WAT? Strings are binaries?

When Elixir sees a list of printable ASCII numbers, Elixir will print that as a charlist (literally a list of characters). Charlists are quite common when interfacing with existing Erlang code.

```
iex> [11, 12, 13]
'\v\f\r'

iex> [104, 101, 108, 108, 111]
'hello'

iex> 'hello' == "hello"
false
```

Booleans

Yes you guessed it right, boolean is an atom. Sometimes instead of using it you can use an atom eg in function return.

```
iex(1)> true
true

iex(2)> true == false
false

iex(3)> is_atom(false)
true
```

```
iex(4)> true and false
false
```

Atoms

An atom is a constant whose name is its own value. Some other languages call the atoms, symbols.

```
iex(1)> :hello
:hello

iex(2)> :hello == :world
false

iex(3)> is_atom(:hello)
true
```

Lists

List are linked list data structure for storing smaller amount of items . You can manipulate it using pattern matching or `Enum` module.

```
iex(1)> [1, 2, true, 3]
[1, 2, true, 3]
iex(2)> [1, 2, 3] ++ [4, 5, 6]
[1, 2, 3, 4, 5, 6]
iex(3)> [true, 2, false] -- [true, false]
[2]
iex(4)> Enum.count([1, 2, 3])
3
iex(5)> Enum.filter([1,2,3], fn n -> n > 1
end)
```

```
iex(6)> list = [1, 2, 3]
iex(7)> hd(list)
1
iex(8)> tl(list)
[2, 3]

# our friend pattern matching
iex(9)[head | rest] = [1, 2, 3]
iex(q0) rest
[2,3]
```

Tuples

Tuple is a simple compound data structure with defined number of elements. It is useful for pattern matching and labeling data.

```
iex(1)> result = {:ok, "hello"}
{:ok, "hello"}

iex(2)> tuple_size(result)
2

iex(3)> elem(result, 1)
"hello"
```

It is also used for returning ok `{:ok, data}` or errors `{:error, reason}` function.

Keyword lists

Keyword list is actually a list of tuples. It looks like dictionary but it is not as it allow item with duplicate keys.

```
# list of tuples
iex(1)> list = [{:a, 1}, {:b, 2}]
[a: 1, b: 2]

# shorter version of the same
iex(2)> list = [a: 1, b: 2]
[a: 1, b: 2]
```

```
iex(3)> Keyword.merge(list, [c: 3])
[a: 1, b: 2, c: 3]

iex(3)> Keyword.update!(list, :a, &(&1 * 2))
[a: 2, b: 2]

iex(5)> list[:a]
1
```

Maps

Maps are the “go to” key-value data structure in Elixir. It is quite handy structure that is simple key/value based and provides better performance characteristics than `Lists`.

```
iex(1)> map = %{a: 1, b: 2, c: #{c1: 3}}

iex(2)> Map.fetch(map, :a)
{:ok, 1}

iex(3)> map[:b]
2

iex(4)> map["non_existing_key"]
nil
```

```
# access nested map
iex(5)> get_in(map, [:c, :c1])
3

# pattern matching
iex(6)> %{a: a_value} = m
iex(7)> a_value
1

# another way of declaring map
iex(8)> %{:a => 1}
```

Structs

Structs are extensions built on top of maps that provide compile-time checks and default values. That is quite helpful in land of dynamic language. That said `Structs` are kind of map so be aware of differences.

```
iex(1)> defmodule MeetupUser do
...>   defstruct name: "Julia", age:
27
...> end
iex(2)> user = %MeetupUser{}
%MeetupUser{age: 27, name: "Julia"}
iex(3)> user_jane = %{user | name:
"Jane"}
%MeetupsUser{age: 27, name: "Jane"
iex(4)> user_jane.name
“Jane”
```

```
iex(5)> is_map(user)
true

iex(6)> %MeetupUser{} = %{}

iex(7)> user.__struct__
MeetupUser

iex(8)> Map.keys(user)
[:__struct__, :age, :name]
```

MapSet has its place

A set can contain any type of elements and prevents duplicities.

```
iex(1)> set2 = MapSet.new([2, 3])

iex(2)> set2 = MapSet.put(set2, 4) |> MapSet.put(4)
#MapSet<2, 3, 4>

iex(3)> MapSet.difference(MapSet.new([1, 2]), set2)
#MapSet<[1]>

iex(4)> MapSet.intersection(MapSet.new([1, 2]), set2)
#MapSet<[2]>

iex(5)> match?(%MapSet{}, MapSet.new())
true
```

Comparison of basic data structures

Elixir	Python
`integer, float`	`Int, Long, Float, Complex`
`true/false`	`True/False`
`atom, eg. :ok`	`-`
`list, eg [1, 2, "string", 3, false]`	`list, eg. [1, 2, "string", 3, False]`
`tuple, eg. {1, "hello"}`	`tuple (1, "hello")`
`map, eg.  %{a: 1, b: 2}`	`dictionary eg. {"a": 1, "b": 2}`
`MapSet, eg. MapSet.new([2, 3])`	`set eg. {2, 3}`

Comparison of basic data structures

Elixir	Python
`Keyword list, eg. [{:a, 1}, {:b, 2}]`	`-`
`Struct, %User{age: 27, name: "Julia"}`	`class or dataclass`
`Pid, #PID<0.89.0>`	`-`
`Function, fn -> :ok end`	`lambda`

Pattern matching

Pattern matching is how our brain works and in functional programming it results in more condense and readable code. Be aware that it is something that once you get it you will miss that in other languages that do not support it.

```
iex(1)> a = 1
1
iex(2)> ^a = 2
** exception error: no match of right
   hand side value 2

iex(3)> 1 = a
true

iex(4)> 2 = a
** (MatchError) no match of right
    hand side value: 1
```

```
iex(5)> [first | rest] = [1, 2, 3]
iex(6)> first
[1]

# you can also ignore some data with _
iex(7)> {a, b, _c} = {1, 2, 3}
{1, 2, 3}
iex(8)> c
** (CompileError) iex:11: undefined function
c/0
```

Power of pattern matching

One of the biggest advantages of pattern matching is that by using them in function signatures you can move decision logic into functions. Based on passed argument function will match...

```
# pattern matching used in functions
def store({:ok, result}) do
   # store result into DB or so...
end
def store({:error, reason}) do
   # unable to read from database
end
def store(unexpectd_result) do
   # log as this is unexpected...
end
```

```
# pattern matching plays nice with
# pipe operator

user_id = 1

user_id
|> read_from_database()
|> do_something()
|> store()
```

Simple conditions

As every other language there is a way to write your decision making logic. Let’s start with usual suspects...

```
meetup = "PyLadies"
if meetup == "PyLadies" do
  IO.puts "PyLadies rock!"
else
  IO.puts "I’m in wrong place :("
end

# Output

PyLadies rock!
:ok
```

```
unless meetup == "PyLadies" do
  IO.puts "I am sad"
else
  IO.puts "Hurrah!"
end

# Output
Hurrah!
:ok
```

With to help with results pattern matching

`With` keyword will help you compose multiple functions by matching it against the pattern on the left side. If the value matches the pattern, with moves on to the next expression. If not you can return error.

```
opts = %{x: 10, y: 15}
with {:ok, x} <- Map.fetch(opts, :y),
     {:ok, y} <- Map.fetch(opts, :x) do
 {:ok, y * x}
end

# Output
{:ok, 150}
```

```
opts = %{x: 10, y: 15}
with {:ok, x} <- Map.fetch(opts, :y),
     {:ok, y} <- Map.fetch(opts, :z) do
 {:ok, y * x}
else
 :error ->
   {:error, :wrong_key}
end

# Output
{:error, wrong_key}
```

Case and its variations...

This is the most simple and very useful way to implement your logic. That said using nested cases is not the best practise as it result in too complicated code.

```
iex(1)> x = 1
iex(2)> case x > 10 do
...>      true -> "Greater than 10"
...>      fale -> "Less then 10"
...>    end
"Less then 10"
```

```
iex(1)> cond do
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   1 + 1 == 2 ->
...>     "But this will"
...> end
"But this will"
```

List comprehension

Comprehensions are syntactic sugar for creating a `Enumerable` based on existing lists.

```
iex(1)> for n <- [1, 2, 3, 4], do: n * n
[1, 4, 9, 16]

iex(2)> values = [good: 1, good: 2, bad: 3, good: 4]
iex(3)> for {:good, n} <- values, do: n * n
[1, 4, 16]

iex(4)> multiple_of_3? = fn(n) -> rem(n, 3) == 0 end
iex(5)> for n <- 0..5, multiple_of_3?.(n), do: n * n
[0, 9]
```

Modules is all you need...

To structure our programs we use modules and where we group our functions. These functions can be public or private. Modules can be in namespaces that should match directory structure.

```
iex(1)> defmodule My.Math do
...>    def sum(a, b) do
...>      do_sum(a, b)
...>   end
...>   defp do_sum(a, b), do: a + b
...> end
iex(2)> My.Math.sum(1, 2)

iex(3)> alias My.Math
iex(4)> Math.do_sum(1, 2)
** (UndefinedFunctionError) …
```

```
# get information about any type
iex(5)> i Math
Term
   My.Math
Data type
   Atom
...
Description
   Call My.Math.module_info() to access
metadata.
Implemented protocols
  IEx.Info, List.Chars, Inspect, String.Chars
```

Error and exception handling ...

Source (https://www.slideshare.net/BrianTroutwine1/let-it-crash-the-erlang-approach-to-building-reliable-services)

Error and exception handling

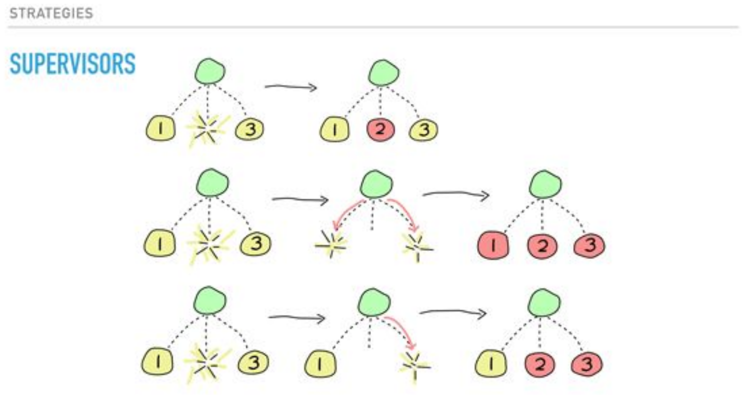
Error handling uses similar tools like try/catch, raising an exceptions. That said BEAM VM philosophy is `Let it crash`. Its meaning is far from what it sounds like.

- Idea is not to spend too much time on defensive programming

Error and exception handling

Error handling uses similar tools like try/catch, raising an exceptions. That said BEAM VM philosophy is `Let it crash`. Its meaning is far from what it sounds like.

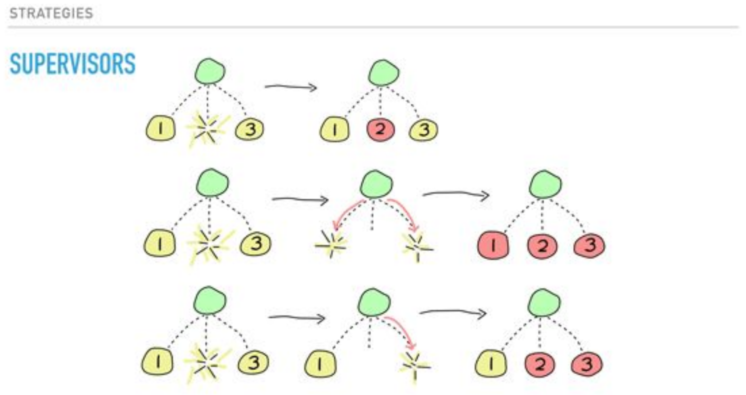
- Idea is not to spend too much time on defensive programming
- Model your program using BEAM VM primitives designed for building fault tolerant systems - **`processes`** and **`supervisors`**



Error and exception handling

Error handling uses similar tools like try/catch, raising an exceptions. That said BEAM VM philosophy is `Let it crash`. Its meaning is far from what it sounds like.

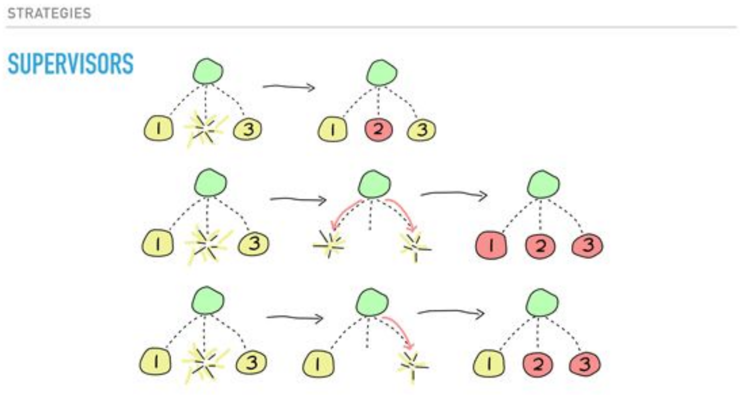
- Idea is not to spend too much time on defensive programming
- Model your program using BEAM VM primitives designed for building fault tolerant systems - **`processes`** and **`supervisors`**
- These primitives help you to design resilient systems



Error and exception handling

Error handling uses similar tools like try/catch, raising an exceptions. That said BEAM VM philosophy is `Let it crash`. Its meaning is far from what it sounds like.

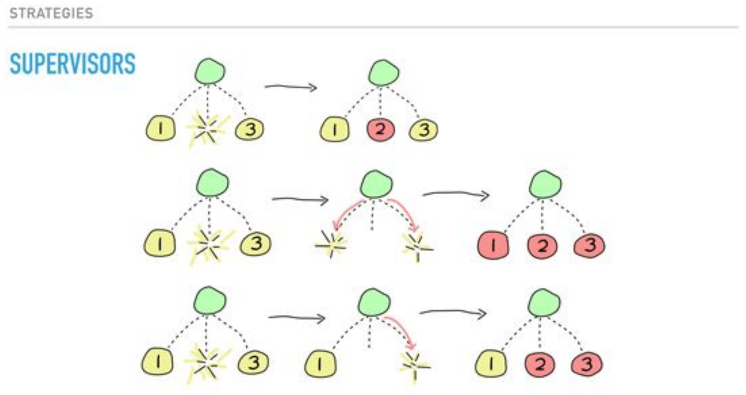
- Idea is not to spend too much time on defensive programming
- Model your program using BEAM VM primitives designed for building fault tolerant systems - **`processes`** and **`supervisors`**
- These primitives help you to design resilient systems
- This does not mean you should not do basic defensive programming checks!



Error and exception handling

Error handling uses similar tools like try/catch, raising an exceptions. That said BEAM VM philosophy is \`Let it crash\`. Its meaning is far from what it sounds like.

- Idea is not to spend too much time on defensive programming
- Model your program using BEAM VM primitives designed for building fault tolerant systems - **`processes`** and **`supervisors`**
- These primitives help you to design resilient systems
- This does not mean you should not do basic defensive programming checks!
- Well known **`try/catch, rescue`** are part of Elixir



There is more language features

- Macros
- Doctests
- Type specs
- Interoperability with other languages (Ports, NIFs, …)

We covered Elixir as a language but there are more interesting features coming from BEAM VM that enables Elixir to shine.

Advanced topics

What we did not mentioned yet...

- Processes
- Sending messages (aka Actors) in between processes and nodes
- Supervisors to deal with crashes
- Monitors and links to observer processes
- Protocols and behaviours to define contract
- OTP patterns - behaviours
- Embedded Mnesia database
- Easy way to write distributed systems
- Observability and data transparency

Behaviours

When building larger or extensible programs and libraries it is sometimes good to be able to express how things should look and what shape they should have.

Behaviours (module attribute) provide a way to:

- define a set of functions that have to be implemented by a module
- ensure that a module implements all the functions in that set
- you can use provided ones or define your own

```
defmodule Parser do
  @callback parse(String.t) :: {:ok,
end
…
defmodule JSONParser do
  @behaviour Parser

  def parse(str) do
    {:ok, Jason.decode!(str)
  end
end
```

Protocols

When building larger or extensible programs and libraries it is sometimes good to be able to express how things should look and what shape they should have. Many modules share the same public API.

Protocols (module attributes) provide a way to:

- enable polymorphism in Elixir…
- you can define them on most of types or `Any` type
- same idea as behaviour but this time for data
- combining with `Structs` is way to express you data in a nice and consistent way

```
# define protocol
defprotocol Size do
  @doc "Calculates the size of type"
  def size(data)
end

# implementation only for Map
defimpl Size, for: Map do
  def size(map), do: map_size(map)
end
…
iex> Size.size([1, 2, 3])
** (Protocol.UndefinedError) protocol Size
not implemented for [1, 2, 3]
```

Elixir	Python
+ Functional, concurrent and fail tolerance + Powerful pattern matching + Allows you to expose only public API of your module + BEAM VM - Batteries included (process management, distribution, database, HTTP, SSH, FTP...) + Not so competitive work market (hiring people interested to learn) - Small community - Fewer companies (but growing :-)) - Slow for numeric computation (unless NIFs) - Understanding OTP design principles as there is no silver bullet	+ Multi paradigm programming language + Many available packages (data, web, scientific, finance, etc.) + Readability, easy to learn + Big job market + Big community, support and resources - Competitive job market - Not possible to hide details of implementation from outside - Many concurrency approaches (none universally supported) - No process management - Breaking changes in between version 2 and 3

Let’s code together

Hands on project

Counting words in a sentence

- Clone using GIT or download source - https://github.com/tgrk/pyladies_elixir/tree/master/word_count
- Three different levels of difficulty (Git branches):
  - `master` - project with basic structure but without any logic
  - `hints` - as above but includes some comments about steps
  - `skeletons` - includes skeletons of functions that represent each step
  - `complete` - full solution if you need inspiration (but hey let’s play it should be fun)
- Choose what level you want to start from
- Open your editor or IDE and try to solve this simple problem…
- **Anytime feel free to ask! I am here to help :-)**
- There is no right way to solve this!

How to deal with unexpected results?

Everyday development tips:

- REPL all the things in `iEx`
- Good old printing:
  - `IO.puts "foo=#{:foo}"`
  - `IO.inspect :foo, label:"HERE"`

- And there is also a debugger:

```
require IEx
value = {:some, :erlang, :value}
IEx.pry
```

In case you want to learn more

- https://elixir-lang.org/getting-started/introduction.html
- https://hexdocs.pm/elixir/
- https://elixir-examples.github.io/
- https://elixirforum.com/

Thank you

Questions