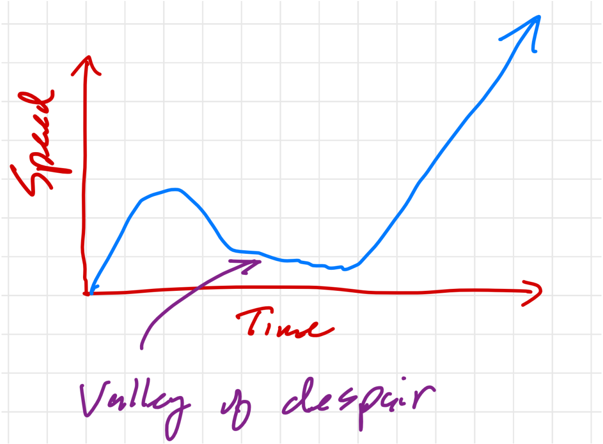
CS 491/691 - Intro to Elixir

Keith Lancaster, Ph.D.

Wait a Tick - Isn’t this a Web Dev Class?

- Why dig into a new language first?
- As you know by now, we are using the Phoenix Framework
    - You can get started *very* quickly using the generators
    - But much as is the case with frameworks like Rails, once you get started building apps, not knowing the language becomes a problem



We will start with *just enough* Elixir to get us going with Phoenix

Background

- Elixir was developed by José Valim, with the first release coming in 2012
- Valim was a Ruby on Rails core developer, so the syntax was heavily influenced by Ruby
- The goal of the project was to create a language that could
  - be used to (easily) build highly concurrent/parallel applications,
  - be fault-tolerant,
  - be easily extensible

Ruby vs Elixir

```ruby
def printme(message)
    puts "My message is #{message}"
end

printme "Hello"
```

```elixir
defmodule Demo do
    def printme(message) do
        IO.puts "My message is #{message}"
    end
end

Demo.printme "Hello"
```

Erlang and the BEAM

- Elixir is a compiled language that runs on the BEAM, the virtual machine that runs Erlang
    - Elixir code compilation involves several steps, resulting in an Erlang module that can utilized by BEAM
- Erlang was developed by Ericcson, a telecom company headquartered in Sweden
- Erlang / Open Telephony Platform (OTP) systems have been running for decades, and are known for extreme reliability
    - This is a requirement, as the systems are used for telephone / communication systems
    - It has been said that if you make a phone call anywhere in Europe, your code has passed through an Erlang system at some point
- Erlang is bundled with OTP, an application operating system and set of libraries
- OTP is *huge*, providing functions to manage concurrency, web and FTP servers, implementations of telecom protocols, etc

Some C

```c
void getLetter(char letter){
    int code = 0;
    switch (letter) {
        case 'A':
            code = 101;
            break;
        case 'B':
            code = 102;
            break;
        default:
            code = 0;
            break;
    }
    printf(("\"%c\" = %d\n"), letter, code);
}
```

An Erlang Service to Add Two Numbers

```erlang
-module(sum_server). -behaviour(gen_server).
-export([   start/0, sum/3,   init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2,   code_change/3 ]).

start() -> gen_server:start(?MODULE, [], []).

sum(Server, A, B) -> gen_server:call(Server, {sum, A, B}).

init(_) -> {ok, undefined}.

handle_call({sum, A, B}, _From, State) -> {reply, A + B, State}.

handle_cast(_Msg, State) -> {noreply, State}.

handle_info(_Info, State) -> {noreply, State}.

terminate(_Reason, _State) -> ok.

code_change(_OldVsn, State, _Extra) -> {ok, State}.
```

The Same Service in Elixir

```
defmodule SumServer do
  use GenServer

  def init(init_arg) do
     {:ok, init_arg}
  end

  def start do
    GenServer.start(__MODULE__, nil);
  end

  def sum(server, a,b) do
    GenServer.call(server,{:sum, a,b});
  end

  def handle_call({:sum, a,b}, _from, state) do
    {:reply, a + b, state}
  end
end
```

Elixir is a *Functional* language

- There are no objects. Functions are grouped into *modules*, but modules are only a way of organizing code
- Variables and data structures are *immutable*
- Pattern matching is used extensively. `if-else` and other imperative constructs are used rarely
    - The use of recursion is common
    - There is a `for` keyword, however it is used in *list comprehensions*.
    - There is no for loop like that in other languages.

Elixir is a *Functional* language (2)

- Function parameters must be sufficient for any calculations
  - For a given set of arguments, the results of a function is always the same.
  - For a particular argument, the function could be replaced by a value and the program result would not change
  - The term for this is *referential transparency*
  - This is possible because function return values are *only a function of the inputs*.
  - Functions that return data from a database will obviously return values based on the current state of the database
- State must be maintained using processes since there are no mutable objects to hold state values

Functional vs OO

- In object-oriented programming, we think in terms of objects that maintain state.
    - State changes (i.e., changes to the attributes of objects) occur in response to *events*
    - **In functional programming, we think in terms of *transforming data***

This is the Way

From a question on Elixir-Lang.org

Instead of asking "how to do X in Elixir", ask "how to solve Y in Elixir"

Typing in Elixir

- Elixir is *dynamically* typed
  - Types are checked at runtime
  - *typespecs* can be used to declare typed function signatures that are used in documentation and by static analysis tools such as *dialyzer*

```
@spec html_escape_to_iodata(String.t()) :: iodata
  def html_escape_to_iodata(data) when is_binary(data) do
    to_iodata(data, 0, data, [])
  end
```

- Elixir can take full advantage of OTP and the Erlang ecosystem
  - The developers of Elixir make it easy to call Erlang functions, and many times do not offer Elixir wrapper functions

Basic Value Types

- Value types:
  - integer: no fixed limit!
  - float: IEEE 754 double precision. There is no separate `double` type
- Atoms
  - Constants that represent the name of something
  - A common use is as keys in a data structures
  - Syntax
    - `:an_atom`
    - Examples:
    - `%{an_atom: 10}` (map) ← can be entered using `%{:an ⇒ "asdf"}`
    - `{:an_atom, 10}` (tuple)

Basic Value Types (2)

- Ranges
    - Defines a set of integers
    - Syntax: `1..10`, `1..10//3` (gives [1,4,7,10] when expanded)
- PID (process IDs)
    - Unique ID for an Elixir (Erlang) process

Strings in Elixir

- Strings are *binaries* in Elixir, i.e., sets of bytes held in contiguous memory
  - For our purposes, we can simply deal with strings as strings and not worry too much about the underlying implementation
  - There is no *native* string type as there is in most languages
  - Strings can be created by surrounding text with double quotes or using binary notation

```
# binaries are printed as strings if the bytes represent printable characters
str = <<65,66,67>> #iex will print "ABC"
```

- Single quoted text is different! These are character lists and are usually only needed when interacting with Erlang functions directly

Collections in Elixir

- Arrays: NOPE
- Lists
  - Lists in Elixir implemented directly on Erlang singly-linked lists.
  - Syntax: `[1,2,3,"aasdf"]`
- Tuples
  - Tuples are ordered collections of values
  - Syntax: `{1,:f,"a string"}`
- Keyword Lists
  - A list containing keyword-value pairs
  - Syntax: `[a: 4, b: 3]`
  - Same as [{:a, 4}, {:b, 3}]

Maps

- Maps are the primary key-value pair construct in Elixir
  - In other languages, they are called dictionaries, hash-maps, etc.

```
iex(1)> m = %{name: "john"}
%{name: "john"}
iex(2)> m.name
"john"
iex(3)> l = %{ m | name: "fred"}
%{name: "fred"}
iex(4)> h = Map.put_new m, :age, 10
%{age: 10, name: "john"}
```

Structs

- Structs are named maps that have fixed members.
- Structs are defined inside a module, and have the same name as the module
- Used to define records for a given type. They can have required and default fields.

```elixir
defmodule MyModule do
    defstruct [:name, :count, age: 10]
# other methods...
end

%MyModule{name: "John", count: 4, age: 20}
```

Pattern Matching

- Pattern matching plays a large part in Elixir applications
- It is used extensively in
  - function parameter lists
  - working with function return values
  - extracting values from data structures

Pattern Matching

- The `=` operator is *not* the assignment operator! It is referred to as the *match operator*
- The operator attempts to match the rhs with the lhs. The lhs is the *pattern*, the rhs is the term to be matched.
- The result of a match expression is always the right-side term you’re matching against
  - `[1,2,3] = [1,2,3]` returns `[1,2,3]`
  - `[1,2,3] = [1,2,4]` returns a match error
- When a variable is present in the lhs it is matched, if possible, with a value from the rhs
  - `[1,2,x] = [1,2,4]` returns `[1,2,4]` with x given the value 4.
- When matching lists, the number of elements must be the same.
  - `[a,b] = [1,2,3]` returns a match error

Pattern Matching Maps and Structs

- Maps and structs *can* be matched with fewer items in the pattern than in the rhs term
    - It is common to have to extract one or more fields from a larger map (struct). *Partial matching* allows for this.
        - `%{age: age} = %{name: "Bob", age: 25}`
- Compound match

```
[_, {name, _}, _] = [{"Bob", 25}, {"Alice", 30}, {"John", 35}]

# Result: name = "Alice"
```

- The "_" means we match against anything. It is referred to as an *anonymous variable* .

Common Use Case - Return Values

- A common idiom for Elixir functions is to return a tuple based on the success or failure of a function
- Pattern matching is used to determine the action based on the response

```elixir
defmodule Patterns do
  def getBoomInfo(message) do
    {:ok,"Go boom and say #{message}"}
  end
end

{:ok, boomer} = Patterns.getBoomInfo("BOOM")
boomer
```

Pattern Matching Strings

- It is not uncommon to need to extract a value that is part of a string.
- Let’s say you have a text string that is read in from a user where the first word is "open", for example "open documents folder"
- If you *know* the format, you can use pattern matching to extract the command using <> , the *string concatenation operator*

```
entered_value = "open documents folder"
"open" <> command = entered_value
IO.puts entered_value # prints "documents folder"
```

Class Exercise

Elixir Basics in LiveBook