Elixir Function Guide

A guide to what you get out of the box in Elixir. This is version 0.2.

Christopher Eyre

Abstract

Elixir Function Guide.

Contents

Elixir Function Guide

Introduction

This book documents the standard libraries that come with Elixir. There are lots of books that cover the extras, this will focus on what you get out of the box. It will also explain some language features as a gentle introduction.

The majority of the book is the function catalogue. I have not found any single place where all this is held, which is why I wrote this book. You can find all this out using the `iex h` command, but it's a slow job.

About the Author

In my spare time I mentor Elixir on exercism.io At the time of writing this (April 2020) I have mentored 6336 solutions and helped 2150 across the Elixir and Groovy tracks.

My degree was in Mathematics where I focused on Mathematical Physics and Game Theory.

I have been working in software development for over 25 years. In that time I have worked in:

- Defence
- Banking
- Futures Trading
- Insurance
- Sports Betting
- Digital Publishing.
- Financial Publishing

Here is my work history:

- Defence Research Agency/Centre for Defence Analysis
- Admiral Computing
- Morgan Stanley Dean Whitter
- Apollo Magazines Ltd

- City Networks
- Munich Re/Watkins Syndicate
- MF Global
- Bet Genius
- Pottermore
- Codurance

My blog https://devrants.blog has been updated (almost monthly) for over a decade and a half. I use it as an external memory.

This is my second book. The first is called Development (https://leanpub.com/development2019). While writing Development I strictly avoided any language specific details, so this book is now a language-specific one.

Why Elixir?

I first found out about Elixir when reading Seven More Languages in Seven Weeks. Several years before I had read the first book Seven Languages in Seven Weeks, and found Erlang to be interesting, providing the software resilience that other languages lack. The downside was the Prolog roots which could make the code hard to read.

Elixir has all the power of Erlang with some semantic sugar, powerful metaprogramming and a very clean syntax.

Installing Elixir

To install Elixir on a mac use `brew install elixir`

`TODO: include instructions for other platforms`

Installed Tools

Elixir has a number of useful utilities supplied:

`elixir` the compiler and run time environment `iex` the interactive elixir environment `mix` the Elixir task runner.

To see that the environment is working try:

```
elixir -v
```

On my machine this returns:

```
Erlang/OTP 21 [erts-10.3.5] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads

Elixir 1.8.2 (compiled with Erlang/OTP 21)
```

Getting Started

Start up iex:

```
Erlang/OTP 21 [erts-10.3.5] [source] [64-bit] [smp:4:4] [ds:4:4:10] [async-threads

Interactive Elixir (1.8.2) - press Ctrl+C to exit (type h() ENTER for help)
iex (1)>
```

You actually need to use Ctrl+C twice to exit iex, although you can use `Ctrl-\`

iex is the basic REPL (Read-Execute-Print-Loop)

You can do simple maths there:

```
iex (1)> 1 + 2
3
```

To get out of iex requires multiple ctrl-c calls or ctrl-|

Functions and Modules

Rather than start with the primitive types and a discussion of immutability I am going to start simply with Functions and Modules

This is the starting piece of any Elixir program. Anything that is not anonymous function must be defined in a module

```
defmodule Sample do
  def hello do
    IO.puts("Hello, World")
  end
end
```

by convention this will be in a file `sample.exs` or `sample.ex` (the language allows you to break this rule, but try to avoid doing so)

`.exs` files are Elixir scripts, the compiled code is not persisted

`ex` file are Elixir source files.

Note that the module is defined with a `defmodule` statement, the name is in PascalCase.

The function is named in `snake_case`. That is entirely lowercase with words seperated by the underscore.

Both of them start the block with a `do` and finish with an `end`

In fact Elixir uses this same pattern for most of the block constructs.

```
defsomething name do
  #Something here
end
```

Not that that explains the defmodule and def statements used above.

Filenames

.ex contains Elixir code that is to be compiled. .exs contains Elixir code that is to be interpreted.

There is no difference in syntax, but the compiled performance will be better.

Naming Conventions

Functions, variables and filenames are named in snake_case Modules are named in PascalCase

```
defmodule PascalCase do

end
```

would be saved in a file called pascal_case.ex

Dots in module names are typically represented as a directory structure.

```
defmodule PacalCase.Core.Server do

end
```

would typically live in lib/pascal_case/core/server.ex

Simple Function Syntax

For one line functions there is a shorter form:

```
defmodule Sample do
  def hello, do: IO.puts("Hello, World")
end
```

Brackets can be optional

If it is unambiguous then brackets can be dropped:

```
defmodule Sample do
  def hello, do: IO.puts "Hello, World"
end
```

Modules can be nested

```
defmodule Outer do
  defmodule Inner do
    def foo, do: 1
  end
end
```

This would be called as `Outer.Inner.foo`

This is identical to:

```
defmodule Outer.Inner do
    def foo, do: 1
end
```

Functions are called by their fully qualified name

The above function can be called from outside the module (say from iex) by calling:

Put the following in a file called Sample.exs

```
defmodule Sample do
  def hello, do: IO.puts "Hello, World"
end
```

From that directory run `iex Sample.exs`

This will leave you at a prompt (ignoring the header)

```
ex(1)>
```

Here you can call the function:

```
iex(1)> Sample.hello
Hello, World
:ok
iex(2)>
```

Here "Hello, World" has been output by the IO.puts function. IO.puts then returns the success code as an atom `:ok`

Elixir functions always return the result of the last expression evaluated. There is no explicit return.

The catalogue

This is a catalogue of the modules and function supplied by Elixir and the modules supplied by Erlang.

These have been captured by using `iex`

The functions are those that can be called unqualified.

Eventually these will be expanded with examples.

It will take time to fully document the erlang modules. Lots of these are the low-level parts of the runtime.

One of the guidelines that Elixir developers follow is not to wrap or reinvent an Erlang function unless you are adding some value.

Modules Starting with A

Module	Description
Access	Key based access to data structures
Agent	Agents are a simple abstraction around state.
Application	A module for working with applications and defining application callbacks. These are the equivalent of runtime libraries in other languages
ArgumentError	Exception. Thrown when no function matches the supplied parameters.
ArithmeticError	Exception thrown when something invalid has been attempted with an Arithmatic operation.
Atom	Convenience functions for working with atoms.

Functions Starting with a

function	Description
abs/1	Returns an integer or float which is the arithmetical absolute value of number.
alias!/1	used in macros to indicate that the given alias should not be explanded.
alias/2	used to set up aliases, often useful with modules names.
and/2	Logical and of the two arguments
apply/2	Invokes the given anonymous function fun with the list of arguments args.
apply/3	Invokes the given function from module with the list of arguments args.

Erlang modules starting with :a

Module	Description
:application	The Erlang version of Application
:application_controller	Controls local applications
:application_master	The root application used by the application_controller
:application_starter	Responsible for processing the `.app` file of a release
:array	Erlang array functions. Provides fixed or expandable arrays.
:atomics	Erlang software lock-free atomic operations
:auth	Erlang network authentication server (This module is deprecated)

Preview

Thanks for reading the preview version of this book.

The full version is available from https://leanpub.com/elixir-function-guide