Elixir notes and guidance and explanation

Contents

1. Introduction . . . 2
2. Guidance and suggestions . . . 2
   2.1. More information . . . 2
   2.2. IEx — interactive mode . . . 2
   2.3. Project management and Mix . . . 3
   2.4. Escripts and executables . . . 4
3. Basic types and how to use them . . . 6
   3.1. Tuples . . . 6
   3.2. Lists . . . 7
   3.3. Char lists . . . 8
   3.4. Strings and bitstrings and binaries . . . 9
   3.5. Sigils . . . 11
   3.6. Iterables/enumerables, streams, etc. . . . 11
   3.7. Keyword lists . . . 12
   3.8. Maps . . . 13
   3.9. Structs . . . 14
4. Syntax and a little semantics . . . 16
   4.1. Modules . . . 16
   4.2. Named functions . . . 17
   4.3. Anonymous functions . . . 17
   4.4. Case, cond, if, and unless . . . 18
5. Doc strings and Function type specs . . . 19
   5.1. Module and function doc strings . . . 19
   5.2. Function type specs . . . 20
6. Metaprogramming and ASTs . . . 21
7. Processes and asynchronous programming . . . 23
   7.1. Agents . . . 25
   7.2. Tasks . . . 25
   7.3. GenServer . . . 25
8. Regular expressions and pattern matching . . . 25
9. Comprehensions . . . 27
10. Protocols . . . 28
11. XML . . . 31
12. Files and input/output . . . 31

13. Database access . . . 32
13.1. Ecto . . . 33
13.2. ETS . . . 33
13.3. DETS . . . 33
13.4. Sqlite3 . . . 33
13.5. PostgreSQL . . . 33
Appendix A: Copyright and License[appendix] . . . 33

1. Introduction

This document is **not** intended to be a tutorial. And, it is certainly **not** a complete reference. It **is** intended to provide notes and reminders of lots of the things that we each need to know as Elixir programmers. Therefore, it has lots of links to Elixir information and has quite a few notes on and examples of common tasks. Hopefully, the table of contents will help you find what you need to know or recall, quickly.

I hope you find this document helpful. And, I hope you enjoy Elixir and are or become as fascinated by it and what it can do and how it does it as I have been.

2. Guidance and suggestions

2.1. More information

- The Elixir Web site: https://elixir-lang.org/docs.html
- Elixir getting started docs: https://elixir-lang.org/getting-started/introduction.html
- Elixir reference documentation: https://elixir-lang.org/docs.html
- Especially useful if you are knowledgeable in Erlang: https://elixir-lang.org/crash-course.html
- Elixir School — https://elixirschool.com/en/

2.2. IEx — interactive mode

`IEx` is your REPL (read evaluate prompt loop) for Elixir. Get help on `IEx` with `h IEx`. Start it with one of the following:

- `iex`
- `iex -S mix` — Use this one if you are in (the directory of) a `Mix` project. It gives you access to the modules defined in that project. And, you can use `iex> IEx.Helpers.recompile` to recompile any modules/files you have edited and modified.

Command history in `IEx` — Elixir `IEx` leans on Erlang for interactive command line history. Enable it with the following environment variable: `ERL_AFLAGS="-kernel shell_history enabled"`. On Linux, I use this `bash` command alias to get that effect:

```
alias iex='ERL_AFLAGS="-kernel shell_history enabled" iex'
```

Configuring `IEx` — Do `h IEx.configuration/0` and `h IEx.configure/1` for help. Use `IEx.configure/1` to customize `IEx`. You can put `IEx` configuration in a file named `.iex.exs` located either in your home directory (global configuration) or your current/project directory (local configuration). Here is an example of an `IEx` configuration file (`.iex.exs`) that removes the command/line count from the `IEx` prompt and asks `IEx` to save a few more lines of history:

```
IEx.configure(
  default_prompt: "%prefix>",
  history_size: 40
)
```

2.3. Project management and Mix

1. Create an new `Mix` project as follows:

   ```
   $ mix new my_new_project
   ```

2. Add dependencies in you project which something like the following:

   ```
   defp deps do
     [
       # {:dep_from_hexpm, "~> 0.3.0"},
       # {:dep_from_git, git: "https://github.com/elixir-lang/my_dep.git", tag:
   "0.1.0"}
       {:sweet_xml, ">0.0.0"},
       {:xmlelixirstructs, github: "dkuhlman/xmlelixirstructs"},
     ]
   end
   ```

   Note that in the above, the line for `sweet_xml` is not actually needed, because `xmlelixirstructs` lists it as a dependency. It's there for purposes of illustration.

3. Retrieve the dependencies and compile them and your project:

   ```
   $ cd my_new_project
   $ mix deps.get
   $ mix deps.compile
   $ mix compile
   ```

4. Start `Iex`, the Elixir interactive shell, in your project:

```
$ iex -S mix
```

In order to get help for `Mix`, do this: `$ mix help`.

Help with `IEx` — the Elixir interactive shell:

- Get a summary of `IEx` and the commands available while in it with this: `iex> h`.
- Get help with `IEx` with the following:

```
iex> h IEx
```

- Tab completion will give you the contents of a module:

```
iex> List.<tab>
Chars                 ascii_printable?/1    ascii_printable?/2
delete/2              delete_at/2           duplicate/2
first/1               flatten/1             flatten/2
foldl/3               foldr/3               improper?/1
insert_at/3           keydelete/3           keyfind/3
keyfind/4             keymember?/3          keyreplace/4
keysort/2             keystore/4            keytake/3
last/1                myers_difference/2    myers_difference/3
pop_at/2              pop_at/3              replace_at/3
starts_with?/2        to_atom/1             to_charlist/1
to_existing_atom/1    to_float/1            to_integer/1
to_integer/2          to_string/1           to_tuple/1
update_at/3           wrap/1                zip/1
```

- While within `IEx`, recompile source code files in your project with the following:

```
iex> IEx.Helpers.recompile
```

2.4. Escripts and executables

See this for information on running Elixir scripts both with and without `Mix`: http://www.davekuhlman.org/elixir-escript-mix.html

An Elixir script (**without** `Mix`) can be as simple as the following:

```
#!/usr/bin/env elixir

defmodule Test do

  def main(args) do
    IO.inspect(args, label: "args")
  end

end

Test.main(System.argv)
```

Notes:

- `System.argv/0` gives us the contents of the command line.

The following example parses the command line and divides the options and arguments into two separate lists:

```
#!/usr/bin/env elixir

defmodule Test09 do
  def test1(args) do
    IO.inspect(args, label: "args")
    {options, arguments, _} = OptionParser.parse(
      args, switches: [key1: :integer, key2: :integer, key3: :integer])
    IO.inspect(options, label: "options")
    IO.inspect(Keyword.keys(options), label: "option keys")
    IO.inspect(Keyword.values(options), label: "option values")
    IO.inspect(arguments, label: "arguments")
    function1 = fn x -> x + Keyword.get(options, :key1) end
    function3 = fn x -> x + Keyword.get(options, :key2) end
    function2 = fn x -> x + Keyword.get(options, :key3) end
    functions = [function1, function2, function3]
    data = [11, 22, 33, 44]
    functions
    |> Enum.each(fn func ->
      result = Enum.map(data, func)
      IO.inspect(result, label: "result")
    end)
  end
end

Test09.test1(System.argv)
```

Notes:

- `OptionParser.parse` separates the options and arguments into lists: options and positional arguments.

- The options are returned as a keyword list. You can use functions in the `Keyword` module to retrieve options and their values.

Here is an example of running the above script:

```
$ elixir test_script.exs --key1 3 --key2 4 --key3 5 aaa bbb ccc ddd
args: ["--key1", "3", "--key2", "4", "--key3", "5", "aaa", "bbb", "ccc", "ddd"]
options: [key1: 3, key2: 4, key3: 5]
option keys: [:key1, :key2, :key3]
option values: [3, 4, 5]
positional arguments: ["aaa", "bbb", "ccc", "ddd"]
result: [14, 25, 36, 47]
result: [16, 27, 38, 49]
result: [15, 26, 37, 48]
```

If the command line for your script is at all complex, consider using the `OptionParser` module in the standard Elixir module library. See:

- https://hexdocs.pm/elixir/OptionParser.html
- In `IEx` do `h OptionParser.parse`, for example.
- A slightly more complex example is in the Escript blog article mentioned above: http://www.davekuhlman.org/elixir-escript-mix.html.

3. Basic types and how to use them

For information on the special characters used to create some of the basic types, see: https://hexdocs.pm/elixir/Kernel.SpecialForms.html

3.1. Tuples

More information:

- https://elixir-lang.org/getting-started/basic-types.html#tuples
- https://hexdocs.pm/elixir/Tuple.html#content

Create a tuple with curly brackets. Examples:

```
iex> t1 = {11, 22, 33}
{11, 22, 33}
iex> t2 = {:birds, ["bluejay", "scrubjay"]}
{:birds, ["bluejay", "scrubjay"]}
```

Get the length of a tuple with `tuple_size/1`:

```
iex> tuple_size(t1)
3
```

Retrieve elements of a tuple by index with `elem/2`. Examples:

```
iex> elem(t1, 0)
11
iex> elem(t1, 1)
22
```

3.2. Lists

More information:

- https://elixir-lang.org/getting-started/basic-types.html#linked-lists
- https://hexdocs.pm/elixir/List.html
- `iex> h List`

A few notes:

- Use `hd/1` and `tl/1` to get the head and tail of a list.
- Pre-pend items to the front of a list with the following syntax:

  ```
  iex> items
  [11, 22, 33, 44]
  iex> [0 | items]
  [0, 11, 22, 33, 44]
  ```

- Use the `++` operator to concatenate two lists.
- Get the length of a list with the `length/1` function.

The `Enum` module is especially helpful with lists (and other objects that satisfy the `Enumerable` protocol, too). In `IEx`, for help, do `h Enum`. Here are examples of the use of `Enum.map/2` and `Enum.each/2`:

```
iex> items = [11, 22, 33, 44]
[11, 22, 33, 44]
iex> items |> Enum.map(fn item -> item * 3 end)
[33, 66, 99, 132]
iex> items |> Enum.each(fn item -> IO.puts("item: #{item}") end)
item: 11
item: 22
item: 33
item: 44
:ok
```

3.3. Char lists

More information:

- https://elixir-lang.org/getting-started/binaries-strings-and-char-lists.html#charlists

Create a char list with single quote marks. Example:

```
iex> fruit = 'apple'
'apple'
iex> fruit
'apple'
```

In contrast, double quotes create a string, and **not** a char list. Example:

```
iex> vegetable = "squash"
"squash"
iex> vegetable
"squash"
```

Convert a string (bitstring) to a char list with either of the following:

```
iex> to_charlist(vegetable)
'tomato'
iex> String.to_charlist(vegetable)
'tomato'
```

Convert a char list to a string with the following:

```
iex> to_string(fruit)
"apple"
```

3.4. Strings and bitstrings and binaries

More information:

- https://elixir-lang.org/getting-started/basic-types.html#strings
- https://elixir-lang.org/getting-started/binaries-strings-and-char-lists.html
- The `<<>>/1` special form: https://hexdocs.pm/elixir/Kernel.SpecialForms.html#%3C%3C%3E%3E/1
- Or, in `IEx`, do: `h <<>>/1`

Create a String with double quotes. Example:

```
iex> vegetable = "squash"
"squash"
iex> vegetable
"squash"
```

Triple double quotes enable us to create multi-line strings:

```
iex> string1 = """
...> my doggie
...> has a long
...> furry tail
...> """
"my doggie\nhas a long\nfurry tail\n"
```

Or, create a string by using bit-string notation. See https://hexdocs.pm/elixir/Kernel.SpecialForms.html#<<>>/1 for help:

```
iex> <<97, 98, 99, 100>>
"abcd"
```

Strings and bit-strings are the same:

```
iex> <<97, 98, 99, 100>> = "abcd"
"abcd"
iex> <<97, 98, 99, 100>> == "abcd"
true
iex> <<97, 98, 99, 100>> === "abcd"
true
```

A technique that we can use to see the bytes in a bit-string is the following:

```
iex> city = "İstanbul"
"İstanbul"
iex> city <> <<0>>
<<196, 176, 115, 116, 97, 110, 98, 117, 108, 0>>
```

We can use a type specifier to create a bit-string containing multi-byte characters. Example:

```
iex> <<304::utf8>>
"İ"
```

Length — Notice that the byte size and character length of strings containing multi-byte characters are different. Examples:

```
iex> city
"İstanbul"
iex> byte_size(city)
9
iex> String.length(city)
8
```

We can use pattern matching on strings/bit-strings:

```
iex> vegetable
"tomato"
iex> <<char::utf8, rest::binary>> = vegetable
"tomato"
iex> char
116
iex> rest
"omato"
```

We can use `String.next_codepoint/1` to iterate over the characters in a string. Examples:

```
iex> String.next_codepoint(vegetable)
{"t", "omato"}
iex> String.next_codepoint(city)
{"İ", "stanbul"}
```

Or, for strings that are a reasonable length, consider converting the string to a `charlist`. Example:

```
iex> city = "İstanbul"
"İstanbul"
iex> String.to_charlist(city)
[304, 115, 116, 97, 110, 98, 117, 108]
iex> city |> String.to_charlist() |> Enum.each(fn char -> IO.puts("character:
#{<<char::utf8>>}") end)
character: İ
character: s
character: t
character: a
character: n
character: b
character: u
character: l
:ok
```

3.5. Sigils

More information:

- https://elixir-lang.org/getting-started/sigils.html
- Visit https://hexdocs.pm/elixir/Kernel.html, Then search the page for "sigil".

3.6. Iterables/enumerables, streams, etc.

There is **no** loop statement in Elixir.

For iteration and "looping", we use recursion (recursive functions) and functions in the `Enum` and `Stream` modules.

Here is an example of a recursive function that applies a function to each item of a List. It's similar to the `Enum.map/2` function.

```
@doc """
Iterate over a list and apply a function to each item, accumulating a result.

## Examples:

    iex> my_list = [11, 22, 33, 44]
    [11, 22, 33, 44]
    iex> func = fn item -> item + 2 end
    #Function<7.126501267/1 in :erl_eval.expr/5>
    iex> Test23.map_list_items(my_list, func, [])
    {:ok, [13, 24, 35, 46]}

"""
@spec map_list_items(List.t(), (any() -> any()), any()) :: {:ok, List.t()}
def map_list_items([], _, acc) do
  {:ok, Enum.reverse(acc)}
end
def map_list_items([item | rest], func, acc) do
  map_list_items(rest, func, [func.(item) | acc])
end
```

And, here is an example of its use:

```
iex> my_list = [11, 22, 33, 44]
[11, 22, 33, 44]
iex> func = fn item -> item + 2 end
#Function<7.126501267/1 in :erl_eval.expr/5>
iex> Test23.map_list_items(my_list, func, [])
{:ok, [13, 24, 35, 46]}
```

And, here is a example of using `Enum.map/2` to do the same thing:

```
iex> Enum.map(my_list, func)
[13, 24, 35, 46]
```

3.7. Keyword lists

Get help in `IEx` with `h Keyword`:

> "Keyword lists are lists of two-element tuples, where the first element of the tuple is an atom and the second element can be any value, used mostly to work with optional values."

More information:

- https://elixir-lang.org/getting-started/keywords-and-maps.html#keyword-lists

- https://hexdocs.pm/elixir/Keyword.html
- `iex> h Keyword`

Notes about keyword lists:

- The items in a keyword list are two-tuples.
- The first item in each two-tuple is an atom.
- There is an equivalence convenience form: `[atom1: value1, atom2: value2, ⋯]`
- The `Keyword` module contains functions for work with keyword lists.

A few examples:

```
iex> [{:aa, 11}, {:bb, 22}, {:cc, 33}]
[aa: 11, bb: 22, cc: 33]
iex> [aa: 11, bb: 22, cc: 33]
[aa: 11, bb: 22, cc: 33]
iex> hd [aa: 11, bb: 22, cc: 33]
{:aa, 11}
```

3.8. Maps

More information:

- https://elixir-lang.org/getting-started/keywords-and-maps.html#maps
- https://hexdocs.pm/elixir/Map.html
- Get help in `IEx` with `h Map`:

> Maps are the "go to" key-value data structure in Elixir.
>
> Maps can be created with the %{} syntax, and key-value pairs can be expressed as key ⇒ value:

Notes:

- Define a map with the form: `%{key1 ⇒ value1, key2 ⇒ value2, ⋯}`.
- When all the keys are atoms, there is an equivalent convenience form: `%{atom1: value1, atom2: value2, ⋯}`.
- When the keys are atoms, we can use the dot-notation to reference values: `my_map.key1`.
- The `Map` module provides functions for working with maps.

Examples:

```
iex> %{:aa => 11, :bb => 22}
%{aa: 11, bb: 22}
iex> %{aa: 11, bb: 22}
%{aa: 11, bb: 22}
iex> %{:aa => 11, "abc" => 22}
%{:aa => 11, "abc" => 22}
iex> lookup = %{aa: 11, bb: 22}
%{aa: 11, bb: 22}
iex> lookup.bb
22
iex> Map.get(lookup, :bb)
22
```

3.9. Structs

More information:

- https://elixir-lang.org/getting-started/structs.html
- https://hexdocs.pm/elixir/Kernel.SpecialForms.html#%25/2
- In `IEx`, do: `h defstruct`.

From the docs (`h defstruct`):

> A `Struct` is a tagged map that allows developers to provide default values for keys, tags to be used in polymorphic dispatches and compile time assertions.

Notes:

- Define a `Struct` with `defstruct` in a module whose name becomes the name of the `Struct`.
- Create a `Struct` by using the following syntax:

```
defmodule MyStructName do
  defstruct fieldname1: defaultvalue1, fieldname2: defaultvalue2, ...
end
```

- We can use functions in the `Map` module on a `Struct`, since it's a `Map` underneath. For example:

```
iex> flower1 = %Flower{name: "star_jasmine", description: "fragrant"}
%Flower{description: "fragrant", name: "star_jasmine"}
iex> Map.keys(flower1)
[:__struct__, :description, :name]
iex> Map.values(flower1)
[Flower, "fragrant", "star_jasmine"]
```

- We can use the dot-notation to dereference values of a `Struct`. Examples:

```
iex> flower1.name
"star_jasmine"
iex> flower1.description
"fragrant"
```

We can also add support functions to the structure in which we define a `Struct`. Example:

```
defmodule Flower do
  defstruct [
    name: nil,
    description: "yet to be provided",
    content: [],
  ]

  def show(flower) do
    IO.puts("name: #{flower.name}  description: #{flower.description}")
    IO.inspect(flower.content, label: "content")
    Enum.each(flower.content, fn color ->
      IO.puts("    color: #{color}")
    end)
  end
end
```

Here is a sample function that creates a few instances of our `Struct` and displays them:

```
defmodule TestFlower do

  def test2() do
    poppy = %Flower{name: "Poppy", description: "bright and cheerful"}
    iris = %Flower{name: "Iris", description: "elegant"}
    freesia = %Flower{name: "Fresia", content: ["yellow", "red", ]}
    IO.inspect(poppy, label: "poppy")
    IO.inspect(iris, label: "iris")
    IO.inspect(freesia, label: "freesia")
    IO.puts("")
    flowers = [poppy, iris, freesia]
    flowers |> Enum.each(fn flower ->
      Flower.show(flower)
      IO.puts("-----")
    end)
    {:ok, flowers}
  end

end
```

When we call the above, we see the following:

```
iex> TestFlower.test2
poppy: %Flower{content: [], description: "bright and cheerful", name: "Poppy"}
iris: %Flower{content: [], description: "elegant", name: "Iris"}
freesia: %Flower{
  content: ["yellow", "red"],
  description: "yet to be provided",
  name: "Fresia"
}

name: Poppy  description: bright and cheerful
content: []
-----
name: Iris  description: elegant
content: []
-----
name: Fresia  description: yet to be provided
content: ["yellow", "red"]
    color: yellow
    color: red
-----
{:ok,
 [
   %Flower{content: [], description: "bright and cheerful", name: "Poppy"},
   %Flower{content: [], description: "elegant", name: "Iris"},
   %Flower{
     content: ["yellow", "red"],
     description: "yet to be provided",
     name: "Fresia"
   }
 ]}
```

4. Syntax and a little semantics

4.1. Modules

More information:

- https://elixir-lang.org/getting-started/modules-and-functions.html
- https://elixir-lang.org/getting-started/module-attributes.html
- For help do: `h defmodule`.

An Elixir source code file can define any number of modules.

Define a module with the `defmodule` macro.

A module can have attributes. They are effectively constants at module level.

A module can have documentation. Create documentation by setting the `@moduledoc` attribute. See below for an example. For more on module attributes, see: https://elixir-lang.org/getting-started/module-attributes.html.

Here is a sample module:

```
defmodule SampleModule do
  @moduledoc """
  Documentation for `SampleModule`.
  """

  @doc """
  Apply a function to each grapheme in a string.

  ## Args:

  - `str` -- An input string.

  - `func` -- A function that takes one argument and returns something.

  ## Examples

      Test23.iter_string(a_string, f1)
      Test23.iter_string(a_string, f1, 10)

  """
  @spec iter_string(String.t(), ((String.grapheme()) -> any()), integer()) :: {:ok,
integer()}
  def iter_string(str, func, count \\ 0) do
    item = String.next_codepoint(str)
    {:ok, count1} = iter_string_aux(item, func, count)
    {:ok, count1 - count}
  end

end
```

4.2. Named functions

More information:

- https://elixir-lang.org/getting-started/modules-and-functions.html#named-functions
- In IEx, do: `h def`.

4.3. Anonymous functions

More information:

- https://hexdocs.pm/elixir/Kernel.SpecialForms.html#fn/1
- https://elixir-lang.org/crash-course.html#anonymous-functions
- https://elixirschool.com/en/lessons/basics/functions/#anonymous-functions

We can also create an anonymous function that has multiple clauses. Here is the recursive function example, but this time defined as an anonymous function:

```
@doc """
Use an anonymous function to iterate over list items.

## Examples

    iex> Test23.anon_map_list_items()

"""
@spec anon_map_list_items((integer() -> integer())) :: :ok
def anon_map_list_items(modifier) do
  func = fn
    ([], acc, _) -> {:ok, Enum.reverse(acc)}
    ([item | rest], acc, this_func) -> this_func.(rest, [modifier.(item) | acc],
this_func)
  end
  [11, 22, 33, 44]
  |> func.([], func)
  |> IO.inspect(label: "1. result")
  # Try it again, this time without pipes.
  simple_list = [1, 2, 3, 4]
  new_simple_list = func.(simple_list, [], func)
  IO.inspect(new_simple_list, label: "2. result")
  :ok
end
```

Notes:

- Our anonymous function, defined with `fn`, has two clauses:
  - One clause matches an empty list and terminates execution.
  - The second clause matches a non-empty list, performs an operation on the current (first) item, and makes the recursive call with modified item added to the accumulator.
- Because we are writing a recursive function, we need to call the anonymous function that we are defining. But, we can't do that yet because that function is still in the process of being defined. So, in order to work around this, **after** the anonymous function has been defined, we call it **and** we pass in the function itself as an argument so that it can call itself.

4.4. Case, cond, if, and unless

More Information:

- https://elixir-lang.org/getting-started/case-cond-and-if.html
- In `IEx`, do:
  - `h case`
  - `h cond`
  - `h if`
  - `h unless`
- `case` and `cond` are special forms. See: https://hexdocs.pm/elixir/Kernel.SpecialForms.html#case/2 and https://hexdocs.pm/elixir/Kernel.SpecialForms.html#cond/1.

Notes:

- Variables bound in a clause do not leak to the outer context.
- If you want to "capture" a value determined by a `case`, `cond`, or `if`, or `unless`, return that value as the value of the nested clauses. If will be returned as the value of the entire expression. Example:

```
iex> result = if true, do: :yes, else: :no
:yes
iex> result
:yes
```

- From the docs: When binding variables with the same names as variables in the outer context, the variables in the outer context are not affected.
- If you want to pattern match against an existing variable, you need to use the `^/1` operator.

5. Doc strings and Function type specs

5.1. Module and function doc strings

More information:

- https://hexdocs.pm/elixir/writing-documentation.html#content

Notes:

- We can add doc strings to modules and functions. We can add type specs to functions.
- Write doc strings in the Markdown lightweight markup language.
- Documentation can be accessed and displayed in `IEx` using `h/1`. Examples:

```
iex> h MyModule
iex> h MyModule.some_function/2
```

- See the following for recommendations and guide-lines on writing documentation: https://hexdocs.pm/elixir/writing-documentation.html#recommendations

An example:

```
defmodule MyModule do
  @moduledoc """
  Documentation for `MyModule`.
  """

  @doc """
  Apply a function to each grapheme in a string.

  ## Args:

  - `str` -- An input string.

  - `func` -- A function that takes one argument and returns something.

  ## Examples

      MyModule.iter_string(a_string, f1)
      MyModule.iter_string(a_string, f1, 10)

  """
  @spec iter_string(String.t(), ((String.grapheme()) -> any()), integer()) :: {:ok,
integer()}
  def iter_string(str, func, count \\ 0) do
  o
  o
  o
end
```

5.2. Function type specs

More information:

- https://elixir-lang.org/getting-started/typespecs-and-behaviours.html#types-and-specs

Notes:

- Function type specs are displayed when you use `h` in `IEx`. For example: `h SomeModule.some_function`.
- Add function type spec immediately before a function definition.

Examples:

```
@spec function1(list(), integer()) :: tuple()
@spec function2(Enumerable.t(), (any() -> any())) :: {:ok, list()}
```

Here are some commonly used type specifiers:

- `any()`
- `atom()`
- `boolean()`
- `Enumerable.t()`
- `float()`
- `integer()`
- `list()`
- `String.t()`
- `tuple()`
- `[{atom(), any()}]` — A keyword list.
- `keyword()` — A keyword list.
- `keyword(t)` — A keyword list of type `[{atom(), t}]`.

A much more complete list of basic types can be found here: https://hexdocs.pm/elixir/typespecs.html

An easy way to discover appropriate type specifiers for a given standard type is to get help for a function that takes that type as an argument. Examples:

- Tuple — In `IEx`, do `h Tuple.append`
- List — In `IEx`, do `h List.delete_at`

You can define your own types. Examples:

```
@type tree_identifier :: {integer(), String.t()}
@type lookup_key :: integer()
```

And, you can document your custom types with `@typedoc`. For more information, see: https://elixir-lang.org/getting-started/typespecs-and-behaviours.html#defining-custom-types

6. Metaprogramming and ASTs

More information:

- See this https://elixir-lang.org/getting-started/meta/quote-and-unquote.html and this https://elixir-lang.org/getting-started/meta/macros.html and even this https://elixir-lang.org/getting-started/meta/domain-specific-languages.html.

- https://elixirschool.com/en/lessons/advanced/metaprogramming/
- In `IEx`, do: `h quote/2`, `h Macro.expand_once/2`, `h Macro.expand/2`, etc.

Some things to keep in mind:

- Macros return (and insert) an AST.
- We use `defmacro` to define a macro.
- In Elixir, **lots** of things are macros.

Metaprogramming and macros are central to the implementation of Elixir itself. Consider the following:

```
iex> a = quote do: unless(junk, do: "Hello")
{:unless, [context: Elixir, import: Kernel],
 [{:junk, [], Elixir}, [do: "Hello"]]}
iex> b = Macro.expand_once(a, __ENV__)
{:if, [context: Kernel, import: Kernel],
 [{:junk, [], Elixir}, [do: nil, else: "Hello"]]}
iex> c = Macro.expand_once(b, __ENV__)
{:case, [optimize_boolean: true],
 [
   {:junk, [], Elixir},
   [
     do: [
       {:->, [],
        [
          [
            {:when, [],
             [
               {:x, [counter: -576460752303423263], Kernel},
               {{:., [], [Kernel, :in]}, [],
                [{:x, [counter: -576460752303423263], Kernel}, [false, nil]]}
             ]}
          ],
          "Hello"
        ]},
       {:->, [], [[{:_, [], Kernel}], nil]}
     ]
   ]
 ]}
```

Notes:

- We use `quote do:` to capture the AST (abstract syntax tree) for an `unless` statement.
- The AST for `unless` expands to the AST for an `if` statement. We use the `Macro.expand_once/2` function to perform this expansion.
- The AST for the `if` expands to the AST for `case`.

- And, `case` does not expand any further. It's a "special form". See: https://hexdocs.pm/elixir/Kernel.SpecialForms.html#case/2
- If, in `IEx`, you do `h unless`, `h if`, you will see that they are macros (i.e., defined with `defmacro`). Note that `h case` reports that about `case` also, but it's special; it's a special form

7. Processes and asynchronous programming

A process is implemented by a function. A typical process (function) waits for a message, sends back a result, and, finally, exits.

Here is an example of the implementation of a simple process and the the functions that (1) start it, (2) run it, and (3) stop it:

```
defmodule TestProcesses do

  @doc """
  Start a process.  Return the process ID.

  ## Examples

      iex> pid = TestProcesses.start

  """
  @spec start() :: pid()
  def start() do
    spawn(__MODULE__, :proc1, ["initial data "])
  end

  @doc """
  Stop the process.

  ## Examples

      iex> TestProcesses.stop(pid)

  """
  @spec stop(pid()) :: :ok
  def stop(pid) do
    send(pid, :stop)
    :ok
  end

  @doc """
  Test the process.  Send multiple messages.  Collect and return the responses.

  ## Example
```

```
    iex> TestProcesses.test(pid, [], 10)

"""
@spec test(pid(), any(), integer()) :: {:ok, list(any())} | {:error, String.t()}
def test(pid, data, count \\ 3) do
  if Process.alive?(pid) do
    acc = 1 .. count
    |> Enum.reduce([], fn idx, acc ->
      send(pid, {:one, self(), "#{idx}. #{data}"})
      send(pid, {:two, self(), "#{idx}. #{data}"})
      acc1 = receive do
        response ->
          IO.inspect(response, label: "response")
          [response | acc]
      end
        acc2 = receive do
          response ->
            IO.inspect(response, label: "response")
            [response | acc1]
        end
        acc2
    end)
    {:ok, Enum.reverse(acc)}
  else
    {:error, "process not alive"}
  end
end

@ doc """
A sample process implementation.

## Examples

    spawn(__MODULE__, :proc1, ["initial data "])

"""
@spec proc1(any()) :: :ok
def proc1(data) do
  receive do
    {:one, pid, msg} ->
      data1 = data <> msg
      IO.puts("one received -- data: #{data}  msg: #{msg}")
      send(pid, data1)
      proc1(data1)
    {:two, pid, msg} ->
      data1 = data <> msg
      IO.puts("two received -- data: #{data}  msg: #{msg}")
      send(pid, data1)
      proc1(data1)
    :stop ->
      IO.puts("stop -- data: #{data}")
```

```
        :ok
    end
  end

end
```

And, here is an example of the use of the above process:

```
iex> pid = TestProcesses.start
iex> TestProcesses.test(pid, [], 5)
iex> TestProcesses.stop(pid)
```

7.1. Agents

More information:

- https://elixir-lang.org/getting-started/mix-otp/agent.html
- https://hexdocs.pm/elixir/Agent.html
- https://elixirschool.com/en/lessons/advanced/concurrency/#agents

Elixir also has Agents, which enable processes to share and update state.

See the link to Elixir School above for an example of an Agent.

7.2. Tasks

Tasks enable a caller to wait for the completion of a task. Tasks enable us to use concurrent code to compute a value asyncronously. See: https://hexdocs.pm/elixir/Task.html

7.3. GenServer

Elixir has `GenServer`, which is a replacement for Erlang's `gen_server` behavior. See: https://hexdocs.pm/elixir/GenServer.html

8. Regular expressions and pattern matching

Regular expressions enable to pattern match on strings and to search a string for a substring that matches a pattern. We can also extract that substring, or replace it. See:

- https://hexdocs.pm/elixir/Regex.html
- https://elixir-lang.org/getting-started/sigils.html#regular-expressions
- Elixir regular expressions implement "Perl Compatible Regular Expressions". See: http://www.pcre.org/.
- A reasonably helpful regular expression cheat sheet is here: https://www.rexegg.com/regex-

quickstart.html.

We can create (and compile) a regular expression with `Regex.compile/1,2` or with the "r" sigil. Examples:

```
iex> {:ok, p1} = Regex.compile("[rR]over")
{:ok, ~r/[rR]over/}
iex> {:ok, p2} = Regex.compile("rover", "i")
{:ok, ~r/rover/i}
iex> p3 = ~r/rover/i
~r/rover/i
```

Using `Regex.compile` can be useful, for example, when you want to create a regular expression from a string in a variable rather than from a literal.

See the `Regex` module for functions that help with regular expressions. Examples:

- Replace substring:

  ```
  iex> Regex.scan(p2, "Hello Rover.  Good doggie, rover.")
  [["Rover"], ["rover"]]
  iex(28)> Regex.replace(p2, "Hello Rover.  Good doggie, rover.", "Curley")
  "Hello Curley.  Good doggie, Curley."
  ```

- Find all substrings. Get the index (location) and length of each substring that is found:

  ```
  iex(23)> a = "ab*c"
  "ab*c"
  iex> {:ok, p1} = Regex.compile(a, "")
  {:ok, ~r/ab*c/}
  iex> Regex.scan(p1, "zxczxcvabbcqwerqerabbbbbcoiuoiu")
  [["abbc"], ["abbbbbc"]]
  iex> Regex.scan(p1, "zxczxcvabbcqwerqerabbbbbcoiuoiu", return: :index)
  [[{7, 4}], [{18, 7}]]
  ```

- Split a string based on a pattern. The substrings matched by the pattern are the locations where the string is split:

  ```
  iex> Regex.split(~r"[./ ]+", "asdf./qwer/ zxcv ouoiu")
  ["asdf", "qwer", "zxcv", "ouoiu"]
  iex> Regex.split(~r"[./ ]+", "asdf./qwer/ zxcv ouoiu", include_captures: true)
  ["asdf", "./", "qwer", "/ ", "zxcv", " ", "ouoiu"]
  ```

- Capture substrings in a map, enabling us to reference the captured substrings by name. Examples:

```
iex> matches = Regex.named_captures(~r/aa(?<matchb>bb)c*(?<matchd>dd)ee/,
"aabbccddee")
%{"matchb" => "bb", "matchd" => "dd"}
iex> matches["matchb"]
"bb"
iex> matches["matchd"]
"dd"
```

9. Comprehensions

More info:

- https://elixir-lang.org/getting-started/comprehensions.html
- https://elixirschool.com/en/lessons/basics/comprehensions/

Notes:

- Comprehensions work on (take as a source) any enumerable. See: https://hexdocs.pm/elixir/Enumerable.html#content.
- Comprehensions are often used to create a list. But, see below for other uses.

Examples:

```
iex> a = [11,22,33,44]
[11, 22, 33, 44]
iex> for n <- a, do: n * 2
[22, 44, 66, 88]
iex> f1 = fn x -> x * 3 end
#Function<44.96251914/1 in :erl_eval.expr/5>
iex> for n <- a, do: f1.(n)
[33, 66, 99, 132]
```

Notes:

- The source of the comprehension can be any enumerable, for example, a list, a `Stream`, a `MapSet`, etc.
- What follows `do:` can be any expression that returns a value, for example a arithmetic expression, a function call, even a constant.

Comprehensions can produce data structures other than lists. For example, we can produce a `Map` or a `MapSet`:

```
iex> a = [11,22,33,44]
[11, 22, 33, 44]
iex> for n <- a, into: %{}, do: {n * 2, n * 10}
%{22 => 110, 44 => 220, 66 => 330, 88 => 440}
iex> for n <- a, into: MapSet.new(), do: {n * 2, n * 10}
#MapSet<[{22, 110}, {44, 220}, {66, 330}, {88, 440}]>
```

Notes:

- We use `into:` followed by a new, empty collectable to specify the target and data type for the result.
- The target must obey the Elixir `Collectable` protocal.
- We can use `into:` with an empty list, which produces the same result as not using `into:` at all. Example:

```
iex> for n <- a, into: [], do: {n * 2, n * 10}
[{22, 110}, {44, 220}, {66, 330}, {88, 440}]
```

10. Protocols

Information about Elixir protocols is here:

- https://elixir-lang.org/getting-started/protocols.html
- https://hexdocs.pm/elixir/Protocol.html#content
- https://elixirschool.com/en/lessons/advanced/protocols/
- In `IEx`, do `iex> h Protocol`.
- I've also written a note about Elixir protocols at my blog: Using Elixir protocols to dispatch on different Elixir structs.

For an example that uses Elixir protocols to dispatch on different Elixir structs, see: http://davekuhlman.org/elixir-protocol-struct.html.

Elixir `Protocol` support enables us to "inject" additional API into multiple data types. Functions defined in a `Protocol` dispatch on the data type of the first argument. Notice that this makes them convenient to use with the Elixir pipe operator "|>", functions on the right side of this operator take their first argument from the pipe.

Here is an example that makes binaries (strings) and charlists polymorphic with respect to `reduce/3` and `map/2`. protocol. See notes below the sample code:

```
defprotocol EnumerableData do
  @moduledoc """
  Make binaries and charlists polymorphic W.R.T. reduce/3 and map/2.
  """
```

```
  def reduce(data, acc, fun)
  def map(data, fun)
end

defimpl EnumerableData, for: BitString do
  def reduce(data, acc, fun) do
    EnumerableStringImpl.reduce(data, acc, fun)
  end
  def map(data, fun) do
    EnumerableStringImpl.map(data, fun)
  end
end

defimpl EnumerableData, for: List do
  def reduce(data, acc, fun) do
    EnumerableListImpl.reduce(data, acc, fun)
  end
  def map(data, fun) do
    EnumerableListImpl.map(data, fun)
  end
end

defmodule EnumerableStringImpl do
  @moduledoc """
  These functions are intended for internal use only.  Use EnumerableData.
  """

  @type t :: String.t()

  @spec reduce(t(), any(), (any(), any() -> any())) :: any()
  def reduce(data, acc, fun) do
    acc1 = reduce_iter_string(String.next_codepoint(data), acc, fun)
    acc1
  end

  defp reduce_iter_string(nil, acc, _) do
    acc
  end
  defp reduce_iter_string({codepoint, str}, acc, func) do
    acc1 = func.(codepoint, acc)
    reduce_iter_string(String.next_codepoint(str), acc1, func)
  end

  @spec map(t(), (any() -> any())) :: List.t()
  def map(data, fun) do
    acc = map_iter_string(String.next_codepoint(data), [], fun)
    Enum.reverse(acc)
  end

  defp map_iter_string(nil, acc, _) do
    acc
```

```
  end
  defp map_iter_string({codepoint, str}, acc, func) do
    new_char = func.(codepoint)
    acc1 = [new_char | acc]
    map_iter_string(String.next_codepoint(str), acc1, func)
  end

end

defmodule EnumerableListImpl do
  @moduledoc """
  These functions are intended for internal use only.  Use EnumerableData.
  """

  @type t :: charlist()

  @spec reduce(t(), any(), (any(), any() -> any())) :: any()
  def reduce(data, acc, fun) do
    acc1 = Enum.reduce(data, acc, fun)
    acc1
  end

  @spec map(t(), (any() -> any())) :: List.t()
  def map(data, fun) do
    data1 = Enum.map(data, fn x -> <<x>> end)
    acc = Enum.map(data1, fun)
    acc
  end

end
```

Notes:

- We use `defprotocol` to define our protocol and the functions it will make polymorphic.
- We use `defimpl` to map the functions defined in our protocol to an implementation for each data type. For type `Binary` (`charlist`), we map these two functions to an implementation that takes a `Binary` as its first argument. For type `List` (`charlist`), we map these two functions to an implementation that takes a `charlist` for its first argument.
- We provide two modules (`EnumerableStringImpl` and `EnumerableListImpl`). One contains the functions that implement our API for type `Binary` (strings) and the other has functions that implement the API for type `charlist`.

The following is a test of the above in `IEx`. It exhibits polymorphism by using a single API (`EnumerableData.map/2` and `EnumerableData.reduce/3`) on two different data types, specifically a binary (string) and a charlist:

```
iex> EnumerableData.map("abcd", fn x -> String.upcase(x) end)
["A", "B", "C", "D"]
iex> EnumerableData.map('abcd', fn x -> String.upcase(x) end)
["A", "B", "C", "D"]
iex> EnumerableData.reduce("abcd", [], fn (item, acc) -> [{item, String.upcase(item)}
| acc] end)
[{"d", "D"}, {"c", "C"}, {"b", "B"}, {"a", "A"}]
iex> EnumerableData.reduce('abcd', [], fn (item, acc) -> [{item, String.upcase(item)}
| acc] end)
[{"d", "D"}, {"c", "C"}, {"b", "B"}, {"a", "A"}]
```

Notes:

- Dispatching to the appropriate implementation — When we call the functions in our protocol, Elixir uses the name of the function and the data type of the first argument in the call to determine which of our implementations to call: the one for `Binary` or the implementation for `charlist`.

11. XML

`SweetXML` makes processing XML documents quite pleasant. See the following:

- https://github.com/kbrw/sweet_xml
- https://hexdocs.pm/sweet_xml/api-reference.html

You can also take a look at my Blog articles that describe some extensions I've made on top of `SweetXML`:

- http://www.davekuhlman.org/xml-elixir-xmerl-functions.html
- http://www.davekuhlman.org/xml-elixir-structs.html

12. Files and input/output

More information:

- https://elixir-lang.org/getting-started/io-and-the-file-system.html
- https://hexdocs.pm/elixir/File.html#content
- https://hexdocs.pm/elixir/IO.html#content
- In `IEx`, do `h File.open/2`, `h IO.read/2`, etc.

Here is an example of reading a text file:

```
iex> {:ok, infile} = File.open("tmp.txt", [:read, :utf8])
{:ok, #PID<0.475.0>}
iex> line = IO.read(infile, :line)
"Selçuk\n"
iex> line = IO.read(infile, :line)
"İstanbul\n"
iex> line = IO.read(infile, :line)
"€€ €€ €\n"
iex> line = IO.read(infile, :line)
:eof
(search)`': line = IO.read(infile, :line)
iex> File.close infile
:ok
```

Notes:

- We open a file for read/input with `:read`.
- We open the file in text mode (as opposed to binary mode) with the `:utf8` option.
- We read one line at a time with `IO.read/2` using the `:line` option. Read the whole file with the `:all` option.
- And, finally, we close the file with `File.close/1`.

Writing output to a file is similar. Example:

```
iex> {:ok, outfile} = File.open("stuff.txt", [:write, :utf8])
{:ok, #PID<0.514.0>}
iex> IO.write(outfile, "aaa bbb\n")
:ok
iex> IO.write(outfile, "ccc ddd\n")
:ok
iex> IO.write(outfile, "eee fff\n")
:ok
iex> File.close outfile
:ok
```

And, then we can display the file:

```
$ cat stuff.txt
aaa bbb
ccc ddd
eee fff
```

13. Database access

13.1. Ecto

More information:

- https://elixirschool.com/en/lessons/ecto/basics/

13.2. ETS

13.3. DETS

13.4. Sqlite3

13.5. PostgreSQL

[to be continued]

Appendix A: Copyright and License[appendix]

Copyright © 2020 Dave Kuhlman. Free use of this documentation is granted under the terms of the MIT License. See https://opensource.org/licenses/MIT.