Episode 02: Data Types

Learn Elixir (https://www.learnelixir.tv/)

Peyton Seigo

2018-12-06

Episode 02: Data Types

Primitive Types

Type-Checking Functions

Elixir provides type predicates in the form `is_type(value)`.

```
is_atom(:hello) # => true
is_list([1, 2, 3]) # => true
is_map(%{key: "val"}) # => true
```

Numbers

```
# Integers
42
1_000_000

# Floats
42.01
1_000_000.05
```

Atoms

```
:atom
:"Contains spaces: and more!"
nil, true, false
ModuleName
```

- `nil`, `true`, `false`, and a `ModuleName` can be written with a colon (`:`) in front because they are atoms.

Binaries (and Strings)

```
# Equivalence: strings are just binaries!
"hello"
<<104, 101, 108, 108, 111>>

"""
```

```
A multiline string typically used
in documentation.
"""
```

Maps

Syntax: `%{key: value, ...}`

```
episode = %{
 name: "Data types",
 author: "Daniel Berkompas"
}

episode.name # => "Data types"
episode[:author] # => "Daniel Berkompas"
```

- Match operator is used to **bind** a variable to a map.
- Keys can be atoms OR strings.
  - e.g. `episode = %{"name": "value"}`
  - Cannot use `episode.name` syntax. Must use `episode["name"]`.

Tuples

```
me = {"Peyton", 18}
{:ok, "The customer was created."}

# Read
elem(me, 0) # => "Peyton"
elem(me, 1) # => 18

# Write
put_elem(me, 1, 19) # => {"Peyton", 19}
```

- 0-based indexing

Lists

```
list = [1, "Beans", 3, 4]
Enum.at(list, 1) # => "Beans"
```

- Arbitrary size
- Any combination of types
- Elixir lists are immutable head/tail pairs (singly-linked list)
  - **Prepend**: fast, does not change list
  - **Append**: slow, recreates every element

```
# Prepend (add element to front of list)
list = [1, 2, 3] # => [1, 2, 3]
[0 | list] # => [0, 1, 2, 3]

## Append (combine two lists)
list ++ [4] # => [1, 2, 3, 4]

## Insert
index = 1
value = 2
List.insert_at([1, 3, 4], index, value) # => [1, 2, 3, 4]
# a new list, [1, 2], points to the original [3, 4] due to immutability
```

Functions

```
add = fn(a, b) ->
  a + b
end
```

Character Lists

```
'hello'
[104, 101, 108, 108, 111]
```

> Unless you're working with an Erlang library, use Binaries instead!

Functions

Syntax: `fn(args) -> ... end`

```
add = fn(a, b) ->
        a + b
      end
add.(1, 2) # => 3
```

- Also known as *anonymous function*, *lambda*, *function literal*, or *closure*.
- Functions are **first-class values** *(also citizen, type, object, or entity)*: they can be
  1. Created without giving a name
  2. Stored in containers such as a value, variable, or data structure
  3. Used as a parameter or return value

Other types from Erlang

- PIDs
- References
- Records
- Port references

High Level Types

Keyword Lists

```
# Syntactic sugar
attrs = [name: "Peyton Seigo",
         email: "test@example.com"]

# Under the hood
attrs = [{:name, "Peyton Seigo"}
         {:email, "test@example.com"}]

# Can access by label
attrs[:name] # => "Peyton Seigo"
attrs[:email] # => "test@example.com"
```

- Implemented as a *list*.
- Thus, it has difference performance characteristics than maps.

Structs

```
# Syntactic sugar
%Episode{
  name: "Data types",
  author: "Daniel Berkompas"
}

# Under the hood
%{
  __struct__: Episode,
  title: "Data Types",
  author: "Daniel Berkompas"
}
```

- Implemented as a *map*.
- `__struct__` usually points to a module containing functions operating on the struct.

Ranges

```
# Syntactic sugar
0..100

# Under the hood
%Range{
  first: 0,
  last: 100
}
```

Regular Expressions

```
# Syntactic sugar
~r/hello/

# Under the hood
%Regex{
  opts: "",
```

```
  re_pattern: {:re_pattern, <<69, 82, 67, 80, 81, 0, ...>>},
  source: "hello"
}
```

Other High Level Types

- Tasks
- Agents
- Streams
- HashDicts
- HashSets