Episode 04: Functions, Modules and Structs

Summary

- Named functions
- Modules
- Documentation
- How to use other modules
- Structs

Terminology

- **Module attributes**: annotations such as `@moduledoc`, denoted by the `@` symbol

Functions

Anonymous Functions

```
# Normal syntax
anon_add = fn(a, b) ->
             a + b
           end
```

The **Capture Operator** (&) is shorthand for anonymous functions.

```
# Shorthand

anon_add = &(&1 + &2)
greet = &(IO.puts "Hello, " <> &1 <> "!")

# Calling
anon_add.(1, 2) # => 3
greet.("Knuth") # => "Hello, Knuth!"
```

Named Functions

```
def named_add(a, b) do
  a + b
end

named_add(1, 2) # => 3
```

Default Arguments

Make default argument values using \\:

```
def fun(arg \\ :default), do: ...
```

Modules

Nested Modules

```
defmodule Math.Division do
  def divide(a, b) do
    a / b
  end
end

# Above is equivalent to:
defmodule Math do
  defmodule Division do
    def divide(a, b) do
      a / b
    end
  end
end
```

Private Functions

```
defmodule Number do
  def format(number) do
```

```
    format = config[:format]
    # ...
  end

  defp config() do
    # get config here
  end
end

Number.config # undefined function: Number.config/0
```

Using Other Modules

Method	Example
Fully qualified name	`Really.Long.OtherModule.other_function`
Aliasing	See example 1
Importing	See example 2
Delegation	See example 3

Examples

Ex. 1: Aliasing Only need to use last module name.

```
defmmodule MyModule do
  alias Really.Long.OtherModule

  def my_function(args) do
    OtherModule.other_function
  end
end
```

Using `alias Really.Long.OtherModule, as:` NewName does the same thing but you can use NewName instead of `OtherModule`.

Ex. 2: Importing Don't need to write other module name.

```
defmmodule MyModule do
  import Really.Long.OtherModule

  def my_function(args) do
    other_function
  end
end
```

Using `:only` allows you to selectively import functions. Using `:except` imports all functions but those listed.

```
import Really.Long.OtherModule,
       only: [other_function: 1]

import Game.Player,
       except: [dont_want_this: 2]
```

Ex. 3: Delegation Outside world sees that `MyModule` has a function called `my_function`, but the work is delegated to `Really.Long.OtherModule`.

```
defmmodule MyModule do
  defdelegate function(arg1, arg2),
              to: Really.Long.OtherModule,
              as: :my_function
end

MyModule.my_function(1, 2) # Calls OtherModule.function(1, 2)
```

Documentation

@ annotations are called *module attributes*.

- `@moduledoc` for modules, written right
- `@doc` for functions

```
defmodule MyModule do
  @moduledoc """
  Explains the module's behaviour.
```

```
  """

  @doc """
  Explains the function's behaviour.

  ## Parameters

  - param - Short description

  ## Examples

      some_function(param)
      result
  """
  def some_function(param) do
    # ...
  end
end
```

Structs

Syntax: `defstruct attr1: val, attr2: ...`

```
defmodule User do
  defstruct name: nil,
            email: nil
end

%User{}
# => %User{email: nil, name: nil}

%User{name: "Peyton"}
# => %User{email: nil, name: "Peyton"}
```

Under the hood:

```
defmodule User do
  def __struct__() do
    %{__struct__: User, name: nil, email: nil}
end
```

Useful `iex` Commands

Command	Description
`iex file.exs`	Runs `iex` with given script
`c("file.exs, ".")`	Compiles `"file.exs"` and writes bytecode (`.beam` file) to current directory