MIXING IT UP with ELIXIR

Level 1

Citizens of the Unknown

Anonymous Functions

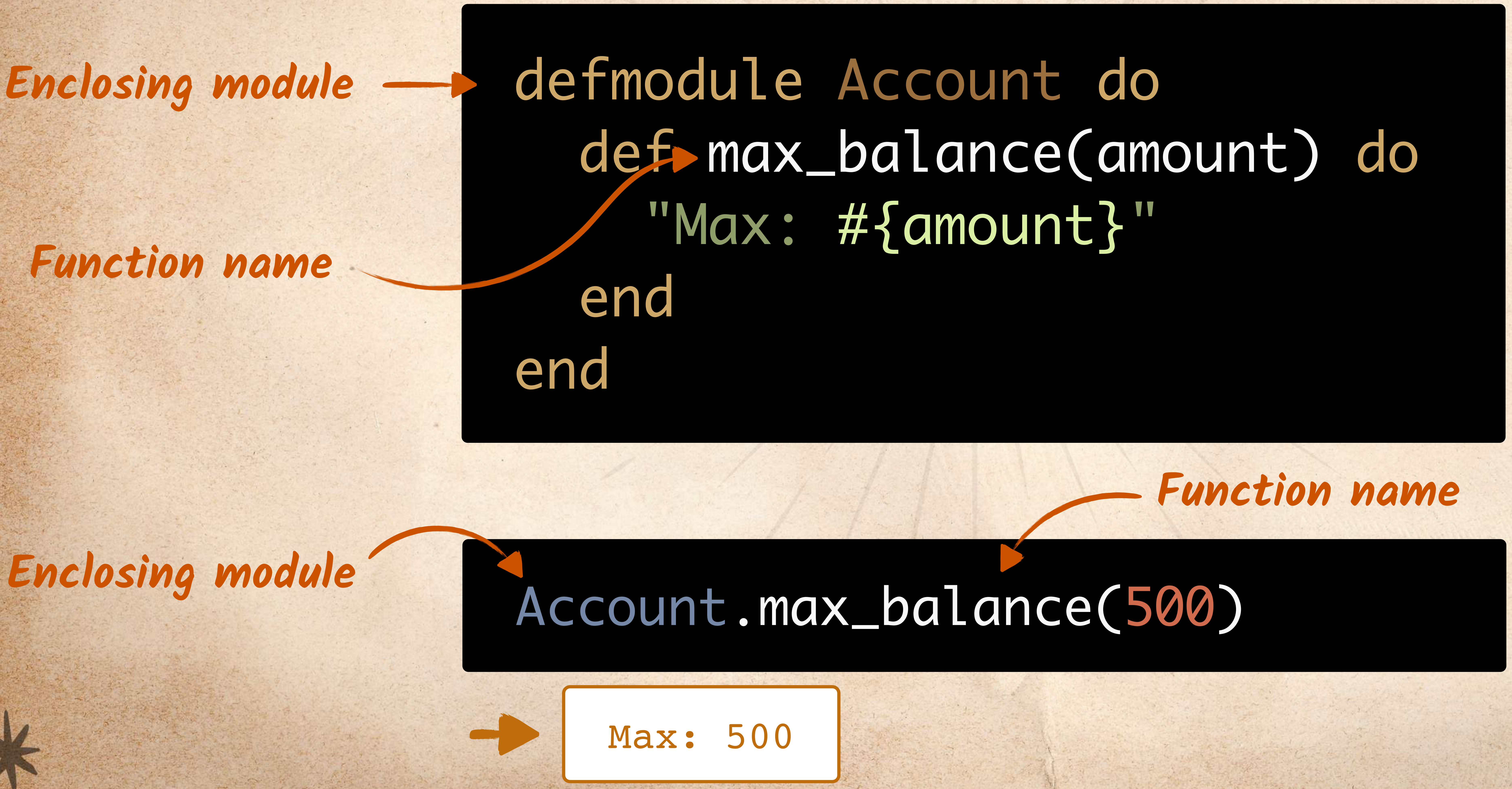
Functions Are First-class Citizens

What does this mean? It means that in Elixir, functions can:

- Be assigned to **variables**
- Be passed around as **arguments** to other functions

What We Know About Named Functions

The functions we've worked with so far have a name and belong to a module.



Enclosing module

Function name

```
defmodule Account do
  def max_balance(amount) do
    "Max: #{amount}"
  end
end
```

Function name

Enclosing module

```
Account.max_balance(500)
```

```
Max: 500
```

No Names, No Modules

Anonymous functions have **no name** and **no modules.** We create them with the `fn ->` syntax.

Single argument

```
max_balance = fn(amount) -> "Max: #{amount}" end
```

Stored in a variable

In order to invoke anonymous functions, we must use the `.( )` syntax.

```
max_balance.(500)
```

```
Max: 500
```

Must use a dot before the parenthesis

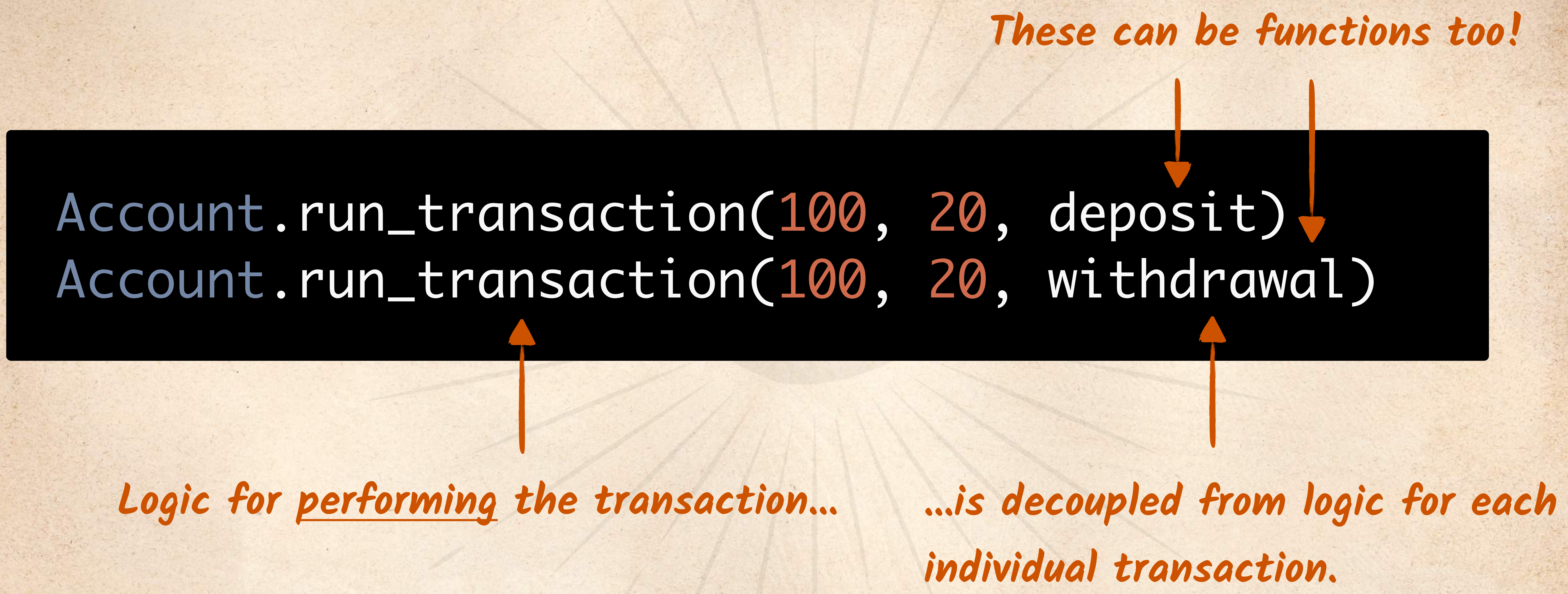
Must pass argument

```
max_balance.( )
```

```
** (BadArityError) #Function<....> with
   arity 1 called with no arguments
```

Decoupling With Anonymous Functions

Named functions can take anonymous functions as arguments. This helps promote **decoupling.**



These can be functions too!

```
Account.run_transaction(100, 20, deposit)
Account.run_transaction(100, 20, withdrawal)
```

Logic for performing the transaction...

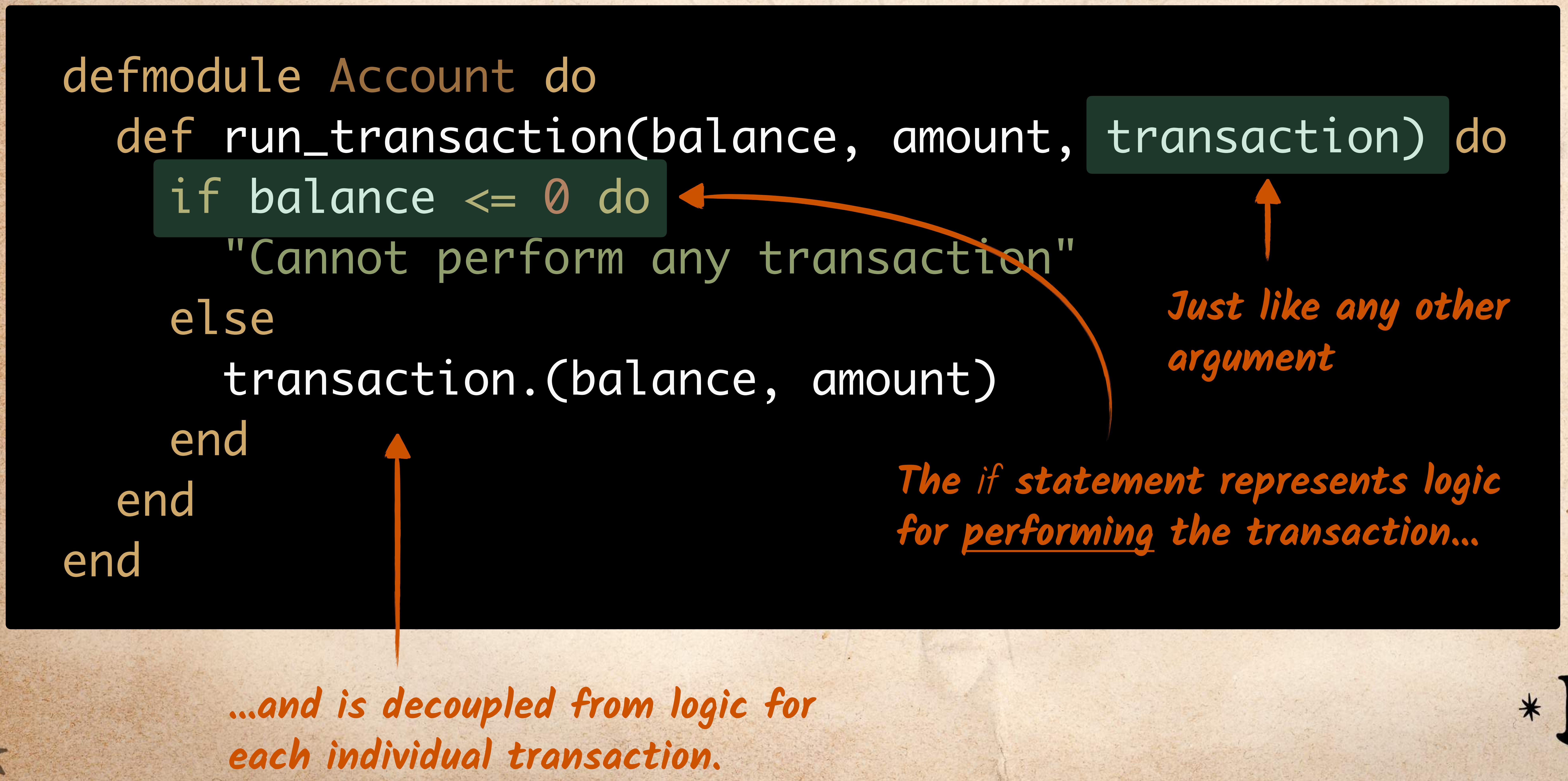
...is decoupled from logic for each individual transaction.

How can we implement this?

* MIXING IT UP *
with
* ELIXIR *

Anonymous Functions as Arguments

The function signature is unchanged, but we must use `.( )` from inside the function body.



```elixir
defmodule Account do
  def run_transaction(balance, amount, transaction) do
    if balance <= 0 do
      "Cannot perform any transaction"
    else
      transaction.(balance, amount)
    end
  end
end
```

Just like any other argument

The if statement represents logic for performing the transaction...

...and is decoupled from logic for each individual transaction.

Passing Anonymous Functions as Arguments

We can pass anonymous functions as arguments, just like with other data types.

```
deposit = fn(balance, amount) -> balance + amount end
withdrawal = fn(balance, amount) -> balance - amount end
```

```
Account.run_transaction(1000, 20, withdrawal)
Account.run_transaction(1000, 20, deposit)
```

```
980
```

```
1020
```

```
Account.run_transaction(0, 20, deposit)
```

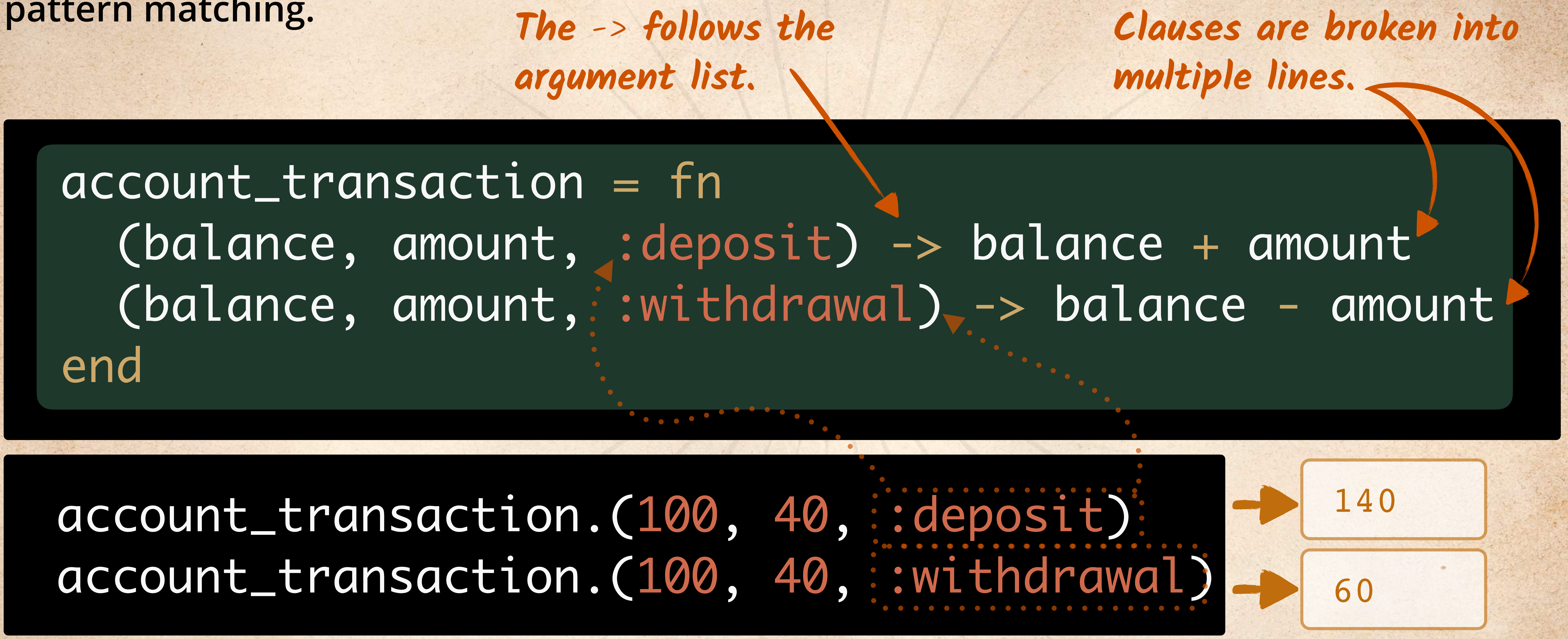
```
Cannot perform any transaction
```

Returns immediately when the balance is 0 — remember?

* MIXING IT UP *
with
* ELIXIR *

Pattern Matching in Anonymous Functions

Similar to named functions, anonymous functions can also be split into **multiple clauses** using pattern matching.



The -> follows the argument list.

Clauses are broken into multiple lines.

```elixir
account_transaction = fn
  (balance, amount, :deposit) -> balance + amount
  (balance, amount, :withdrawal) -> balance - amount
end
```

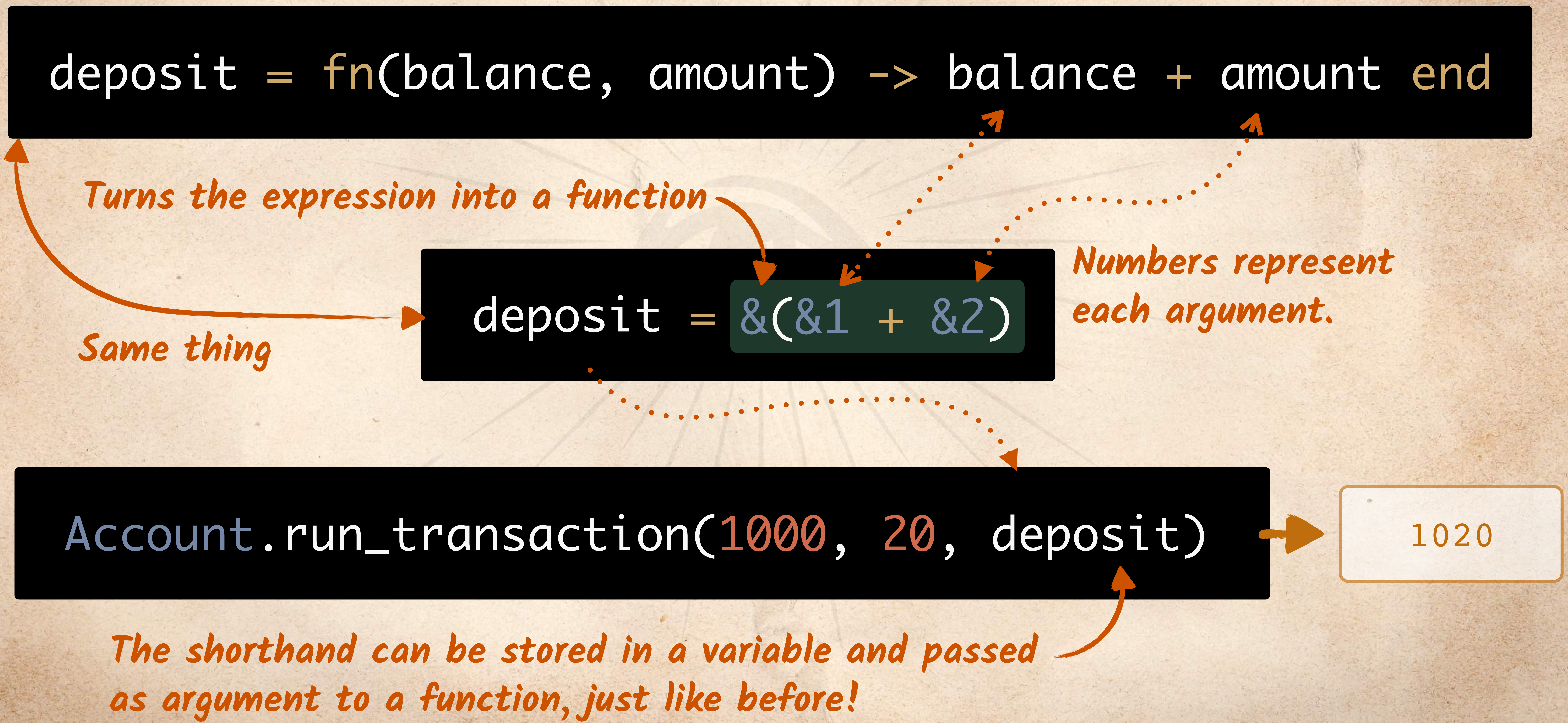
```elixir
account_transaction.(100, 40, :deposit)
account_transaction.(100, 40, :withdrawal)
```

```
140
```

```
60
```

Anonymous Function Shorthand Syntax

The `&` operator is used to create helper functions in a short and concise way.



```
deposit = fn(balance, amount) -> balance + amount end
```

Turns the expression into a function

Numbers represent each argument.

Same thing

```
deposit = &(&1 + &2)
```

```
Account.run_transaction(1000, 20, deposit)
```

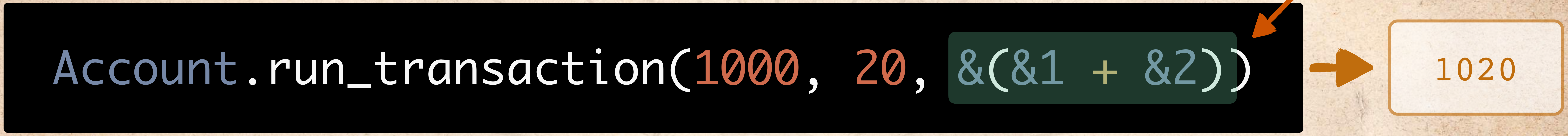
```
1020
```

The shorthand can be stored in a variable and passed as argument to a function, just like before!

Using the Shorthand Inline

The shorthand version of anonymous functions is often found used inline as arguments to other functions.

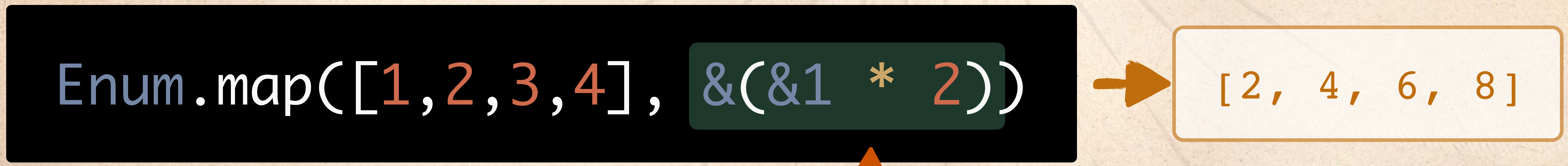
Can be defined inline too!



```
Account.run_transaction(1000, 20, &(&1 + &2))
```

```
1020
```

Enum.map is part of Elixir's standard library. It returns a list where each item is the result of invoking a function on each corresponding item of enumerable.



```
Enum.map([1,2,3,4], &(&1 * 2))
```

```
[2, 4, 6, 8]
```

Shorthand function that multiplies its argument by 2

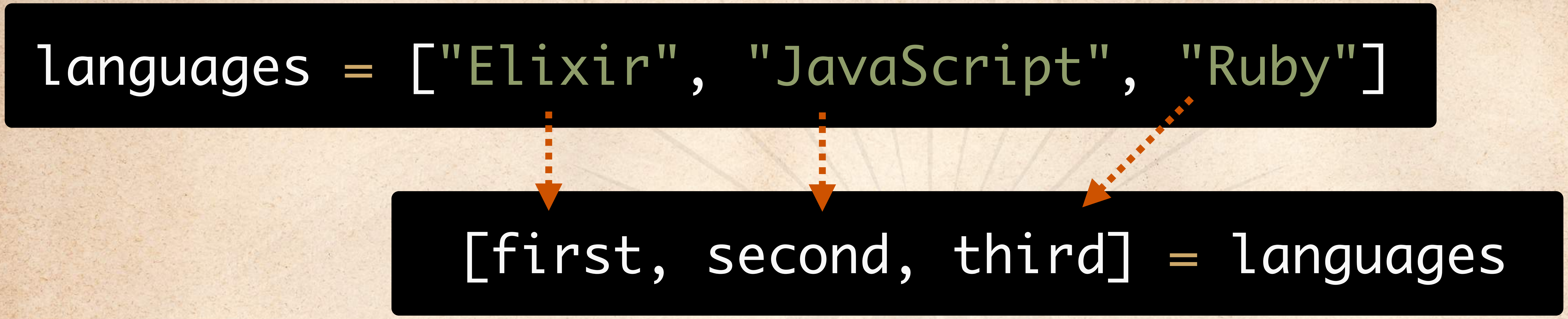
Level 2

The End Is the Beginning

Lists & Recursion

Reading Elements From a List

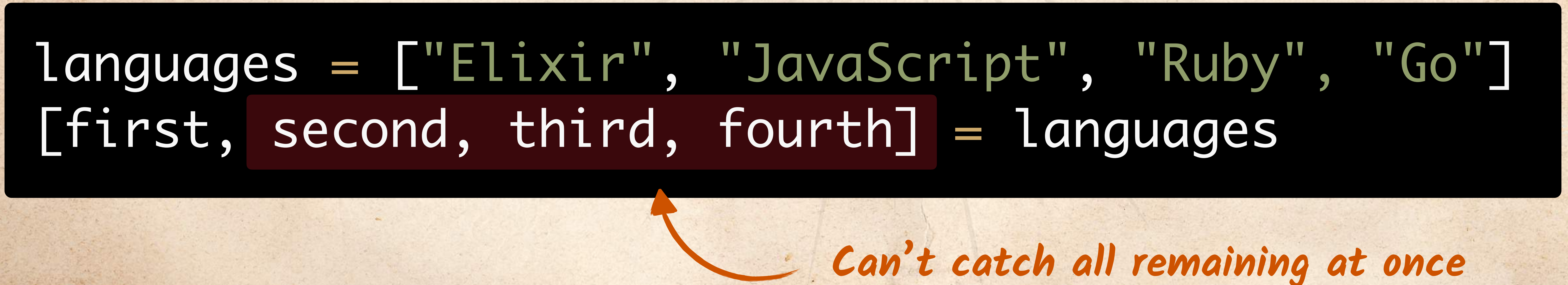
We can use pattern matching on lists to read individual elements.



```
languages = ["Elixir", "JavaScript", "Ruby"]
```

```
[first, second, third] = languages
```

However, this does **not** scale well as the list grows...



```
languages = ["Elixir", "JavaScript", "Ruby", "Go"]
[first, second, third, fourth] = languages
```

Can’t catch all remaining at once

* MIXING IT UP *
with
* ELIXIR *

Splitting a List With the cons Operator

The cons operator `|` is used to split a list into head (first element) and tail (remaining elements).

```
languages = ["Elixir", "JavaScript", "Ruby"]
[head | tail] = languages
```

"Elixir" ["JavaScript", "Ruby"]

Pick the first...

```
languages = ["Elixir", "JavaScript", "Ruby"]
[head | _ ] = languages
```

...and ignore the rest with no compiler warnings.

* MIXING IT UP *
with
* ELIXIR *

Using cons in Function Pattern Matching

The cons operator can be used in function pattern matching to split lists into head and tail.

```
defmodule Language do
  def print_list([head | tail]) do
    IO.puts "Head: #{head}"
    IO.puts "Tail: #{tail}"
  end
end
```

Split single list argument into head and tail

```
Language.print_list(["Elixir", "JavaScript", "Ruby"])
```

```
Head: Elixir
Tail: JavaScriptRuby
```

No for Loops

There are **no** for loops in Elixir. How can we iterate through a list without using a for loop?

```
defmodule Language do
  def print_list([head | tail]) do

     ?????

  end
end
```

Cannot use a loop here

```
Language.print_list(["Elixir", "JavaScript", "Ruby"])
```

👀 *We see this now...*

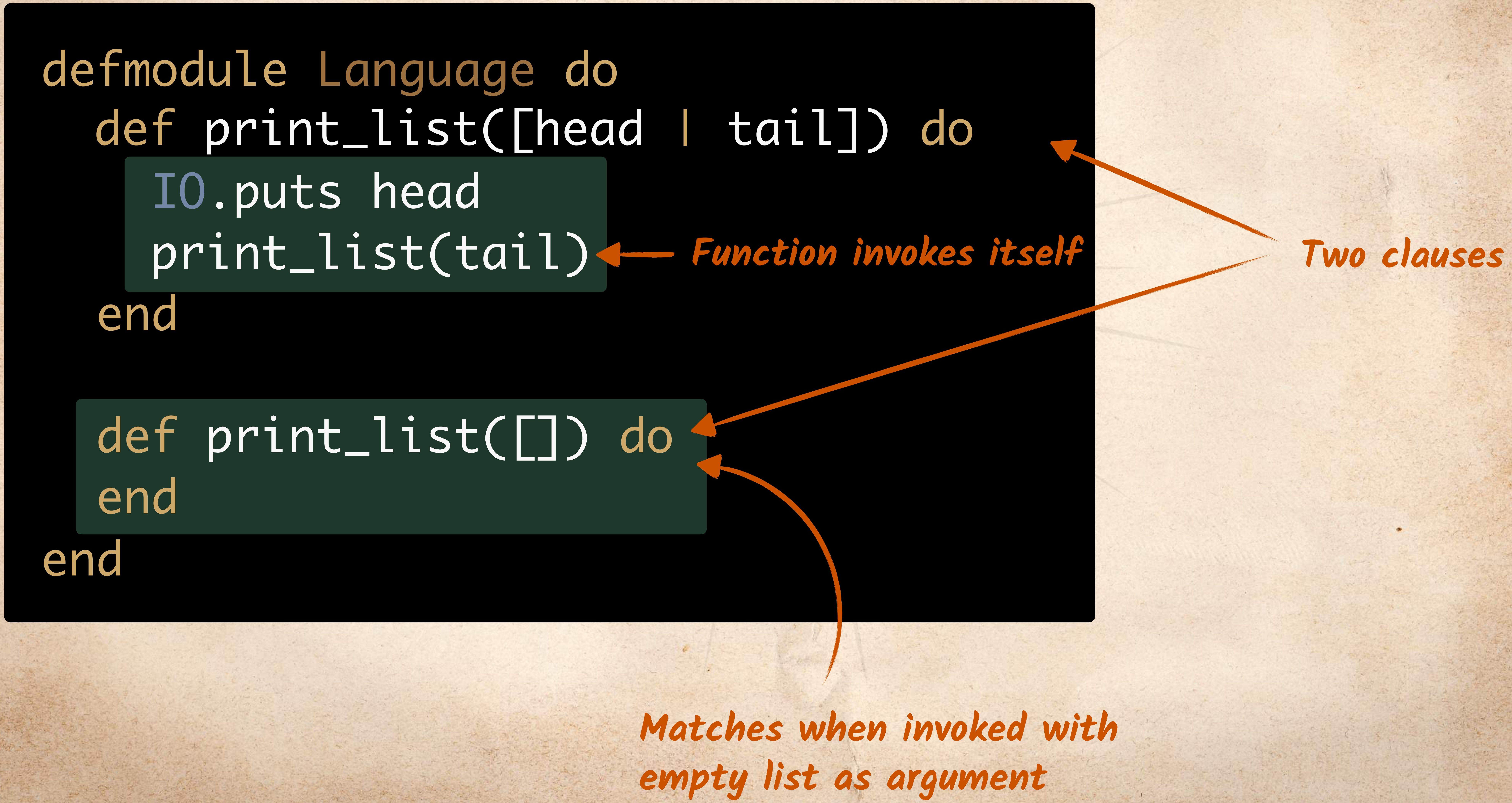
```
Head: Elixir
Tail: JavaScriptRuby
```

...but we want this.

```
Elixir
JavaScript
Ruby
```

Understanding Recursion

Recursive functions are functions that perform operations and then **invoke themselves.**



```
defmodule Language do
  def print_list([head | tail]) do
    IO.puts head
    print_list(tail)
  end

  def print_list([]) do
  end
end
```

Function invokes itself

Two clauses

Matches when invoked with
empty list as argument

Two Cases for Recursion

All recursive functions involve the following two cases (or two **clauses**):

1. The base case, also called **terminating scenario**, where the function does NOT invoke itself.

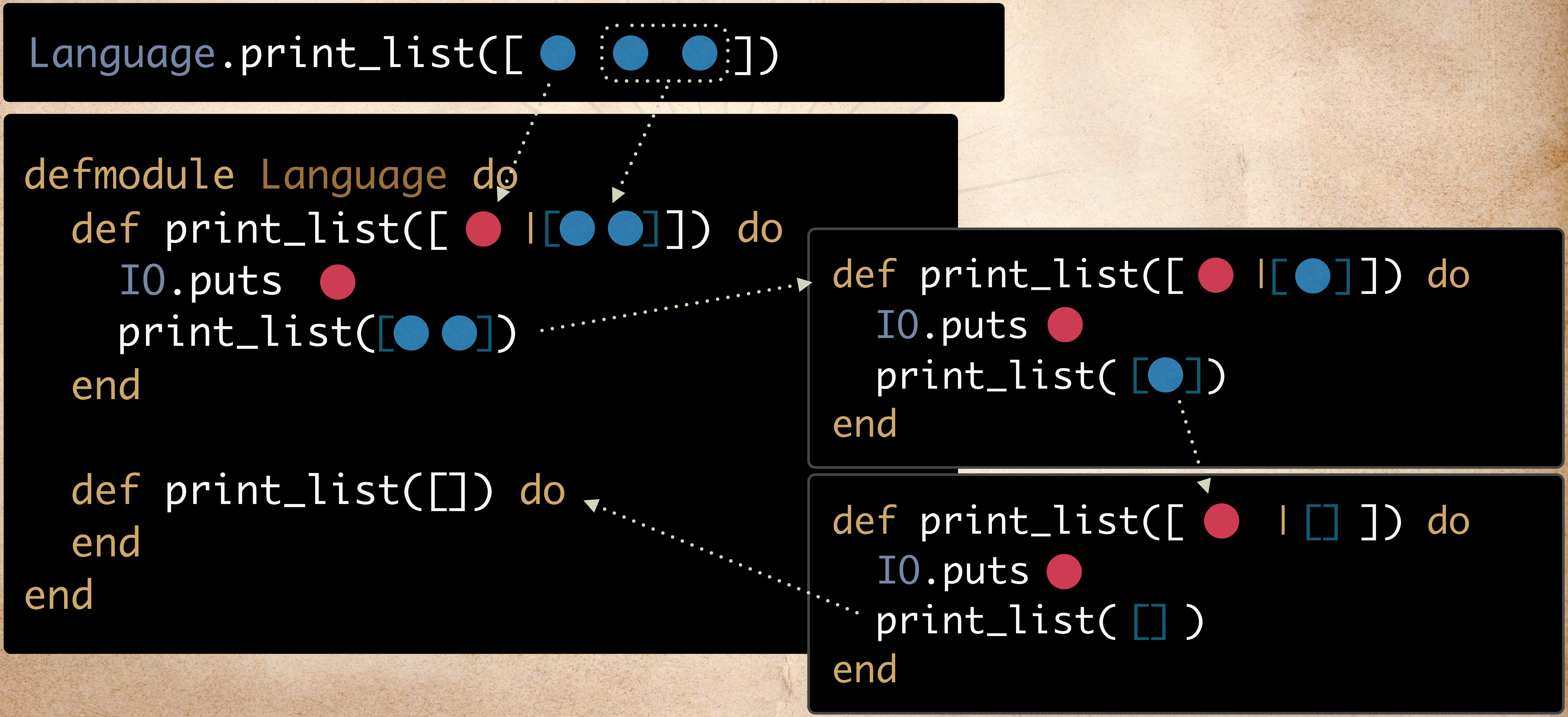
```
def print_list([]) do
end
```

2. The **recursive case**, where computation happens and the function invokes itself.

```
def print_list([head | tail]) do
  IO.puts head
  print_list(tail)
end
```

Loops With Recursion

splitting lists with the cons operator + pattern matching + recursion = loop



```
Language.print_list([● ● ●])
```

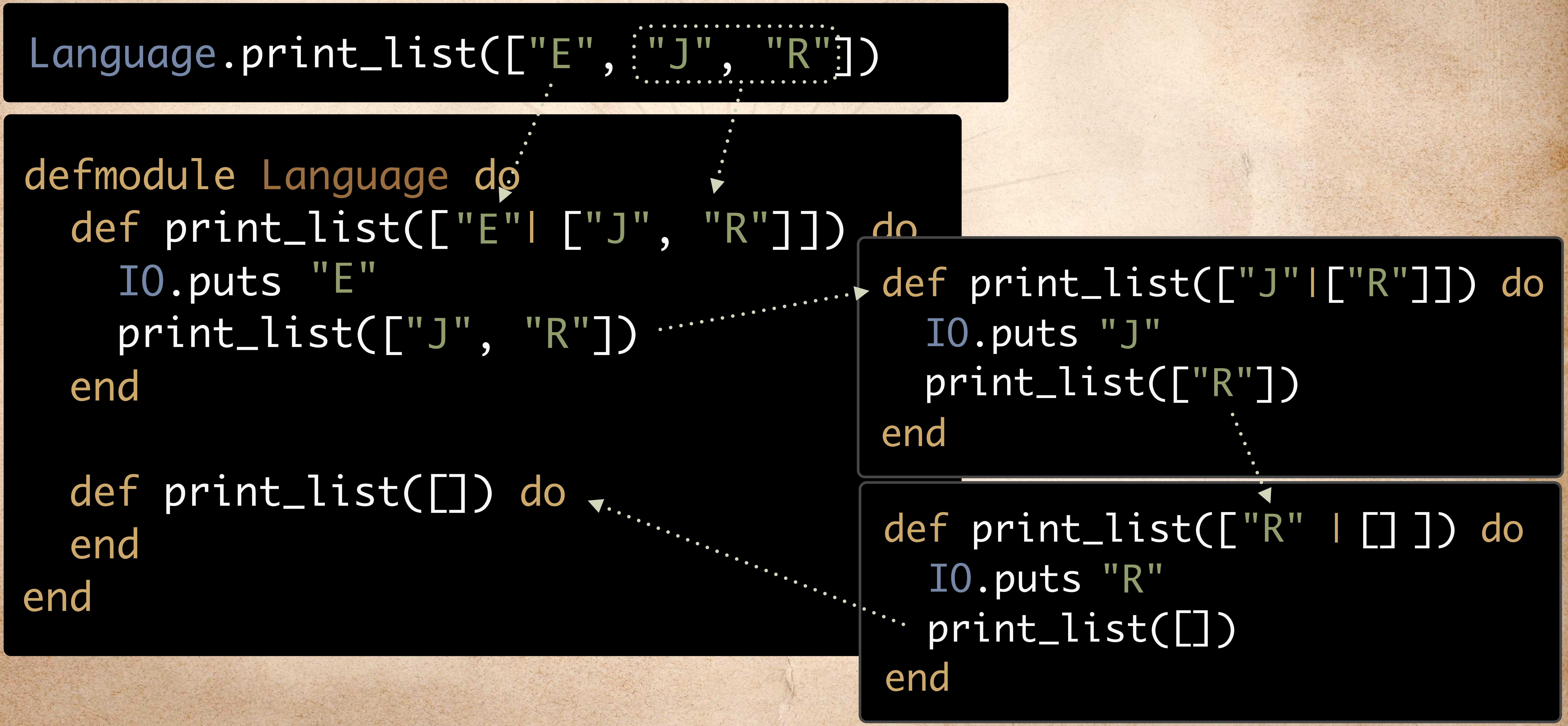
```
defmodule Language do
  def print_list([● |[● ●]]) do
    IO.puts ●
    print_list([● ●])
  end

  def print_list([]) do
  end
end
```

```
def print_list([● |[●]]) do
  IO.puts ●
  print_list( [●])
end
```

```
def print_list([● | [] ]) do
  IO.puts ●
  print_list( [] )
end
```

The Real Step-by-step Recursion Code

The principle of recursion can be applied to any other data types, like strings.



```
Language.print_list(["E", "J", "R"])
```

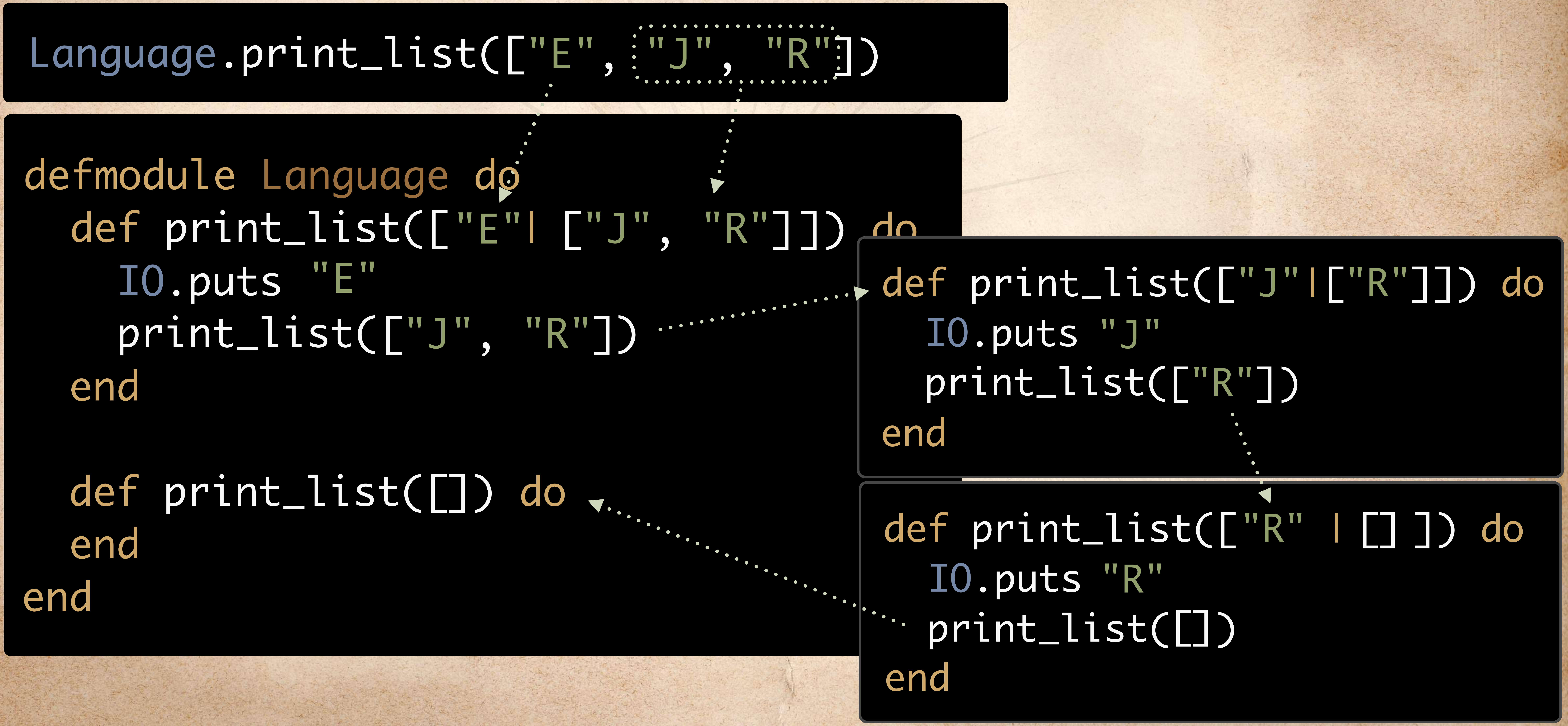
```
defmodule Language do
  def print_list(["E"| ["J", "R"]]) do
    IO.puts "E"
    print_list(["J", "R"])
  end

  def print_list([]) do
  end
end
```

```
def print_list(["J"|["R"]]) do
  IO.puts "J"
  print_list(["R"])
end
```

```
def print_list(["R" | [] ]) do
  IO.puts "R"
  print_list([])
end
```

Loops With Recursion

splitting lists with the cons operator + pattern matching + recursion = loop



```
Language.print_list(["E", "J", "R"])
```

```
defmodule Language do
  def print_list(["E"| ["J", "R"]]) do
    IO.puts "E"
    print_list(["J", "R"])
  end

  def print_list([]) do
  end
end
```

```
def print_list(["J"|["R"]]) do
  IO.puts "J"
  print_list(["R"])
end
```

```
def print_list(["R" | [] ]) do
  IO.puts "R"
  print_list([])
end
```

The Complete Recursive Code

Using recursion, we can now iterate through elements from a list!

```
defmodule Language do
  def print_list([head | tail]) do
    IO.puts head
    print_list(tail)
  end

  def print_list([]) do
  end
end
```

```
Elixir
JavaScript
Ruby
```

```
Language.print_list(["Elixir", "JavaScript", "Ruby"])
```

Level 3–1

Tuples & Maps

Tuples

Creating Tuples

We use curly braces `{}` to represent tuples, an ordered collection of elements typically used as return values from functions.

A valid tuple

```
{:functional, "elixir", 2012}
```

Different data types

Tuples can hold many elements of different data types, but more often than not, we'll work with **two-element** tuples where the first element is an atom.

First element is usually an atom

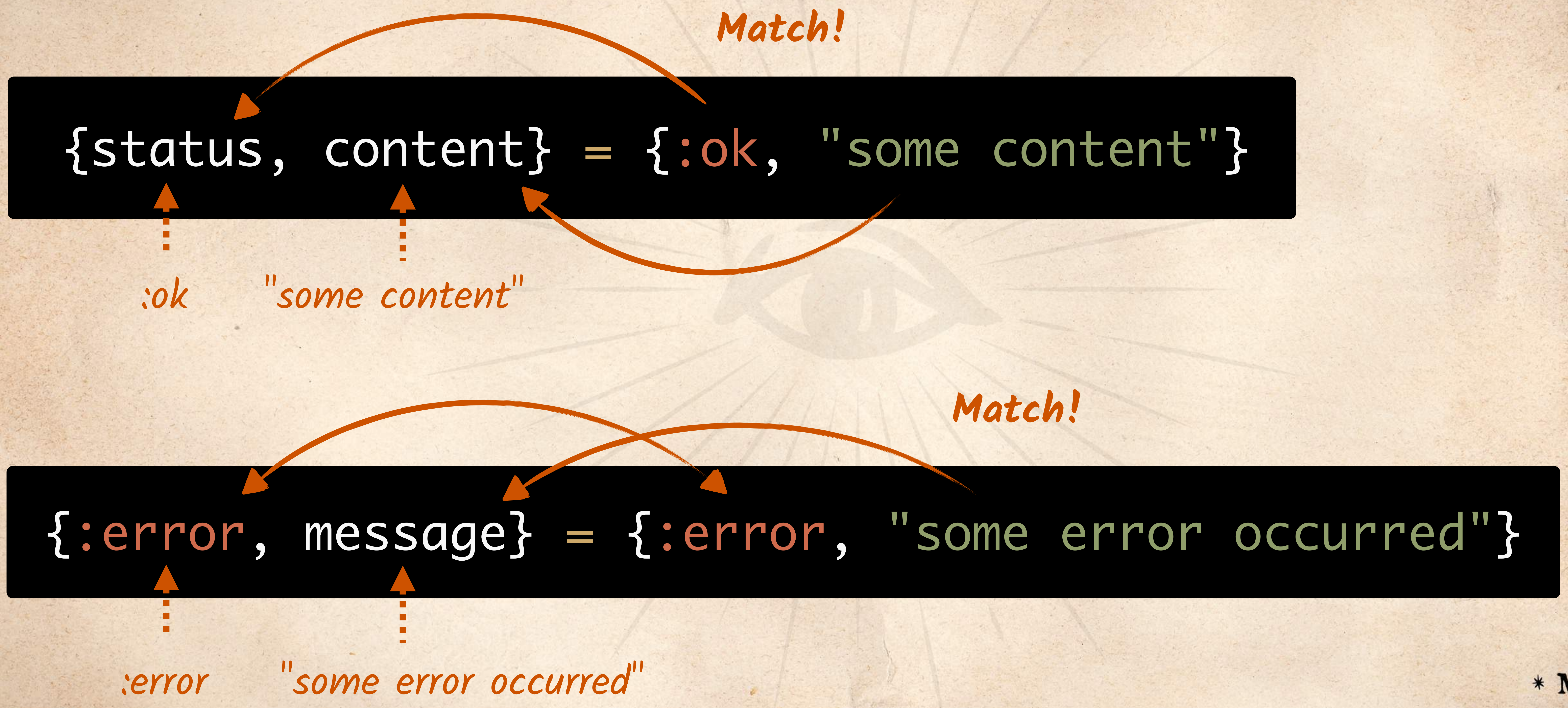
```
{:ok, "some content"}
{:error, :enoent}
```

Data type for second element will vary

atom representing an <u>unknown file</u> error

Tuples & Pattern Matching

We can use **pattern matching** to read elements from tuples.

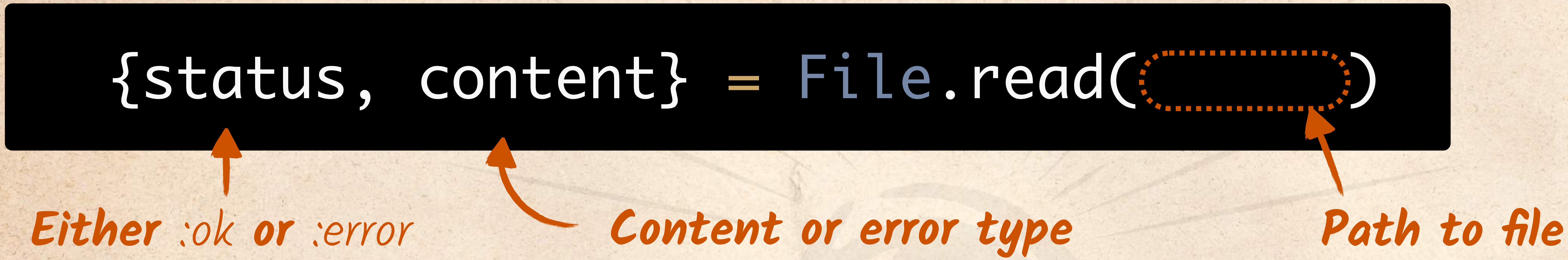


```
{status, content} = {:ok, "some content"}
```

```
{:error, message} = {:error, "some error occurred"}
```

Returning Tuples From Functions

The File.read function from Elixir's standard library returns a tuple with two elements: an atom representing the status of the operation and either the content of the file or the error type.



```
{status, content} = File.read(        )
```

Either :ok or :error

Content or error type

Path to file

```
{:ok, content} = File.read("transactions.csv")
```

```
{:ok, content} = File.read("file-that-doesnt-exist")
```

✗

```
** (MatchError) no match of right hand side value: {:error, :enoent}
```

```
{:error, content} = File.read("file-that-doesnt-exist")
```

✓

Pattern Matching Tuples From Functions

We can pattern match tuples in function arguments to read values passed in function calls.

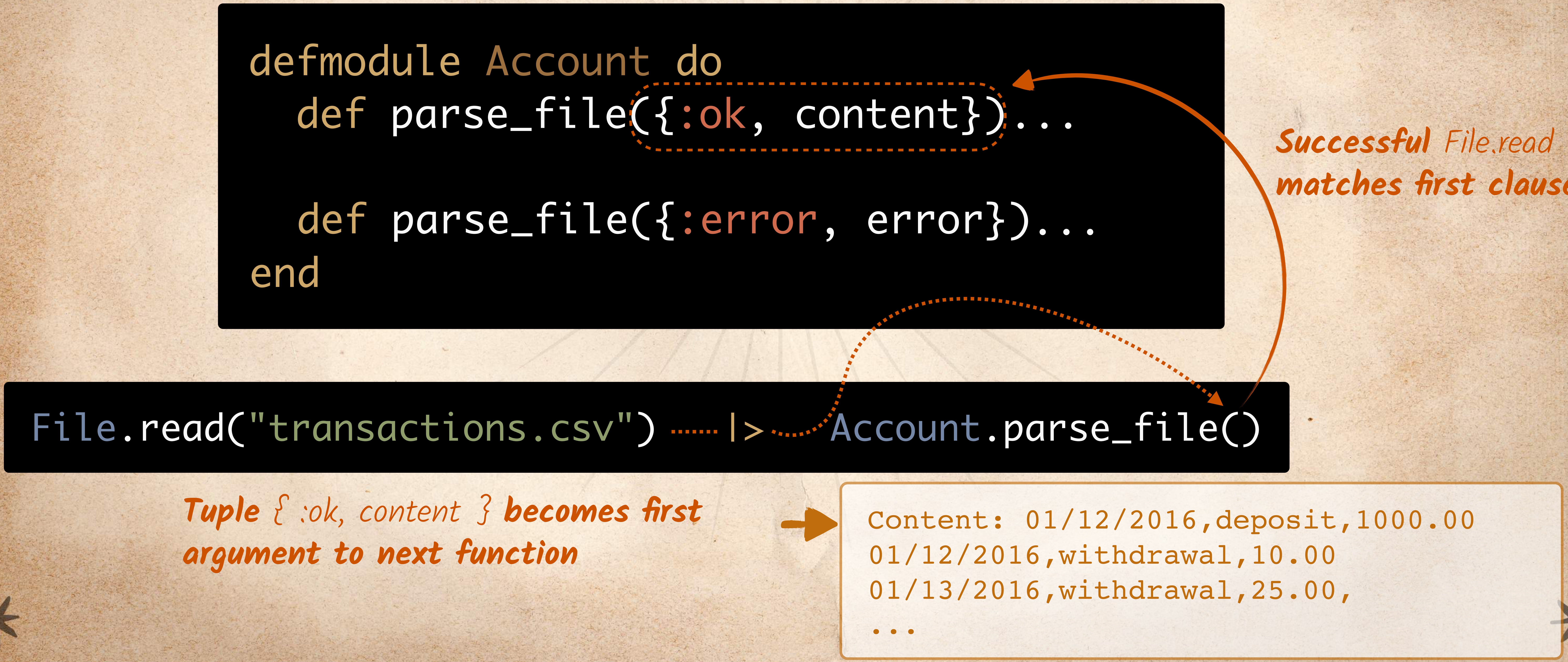
```
defmodule Account do
  def parse_file({:ok, content}) do
    IO.puts "Transactions: #{content}"
  end

  def parse_file({:error, error}) do
    IO.puts "Error parsing file"
  end
end
```

This clause matches a successful File.read operation.

This clause matches an unsuccessful File.read operation.

Matching Successful Return Value

The pipe operator `|>` can be used to pass the result of reading the given file over to the newly created parse_file function from the Account module.



```
defmodule Account do
  def parse_file({:ok, content})...

  def parse_file({:error, error})...
end
```

Successful File.read matches first clause

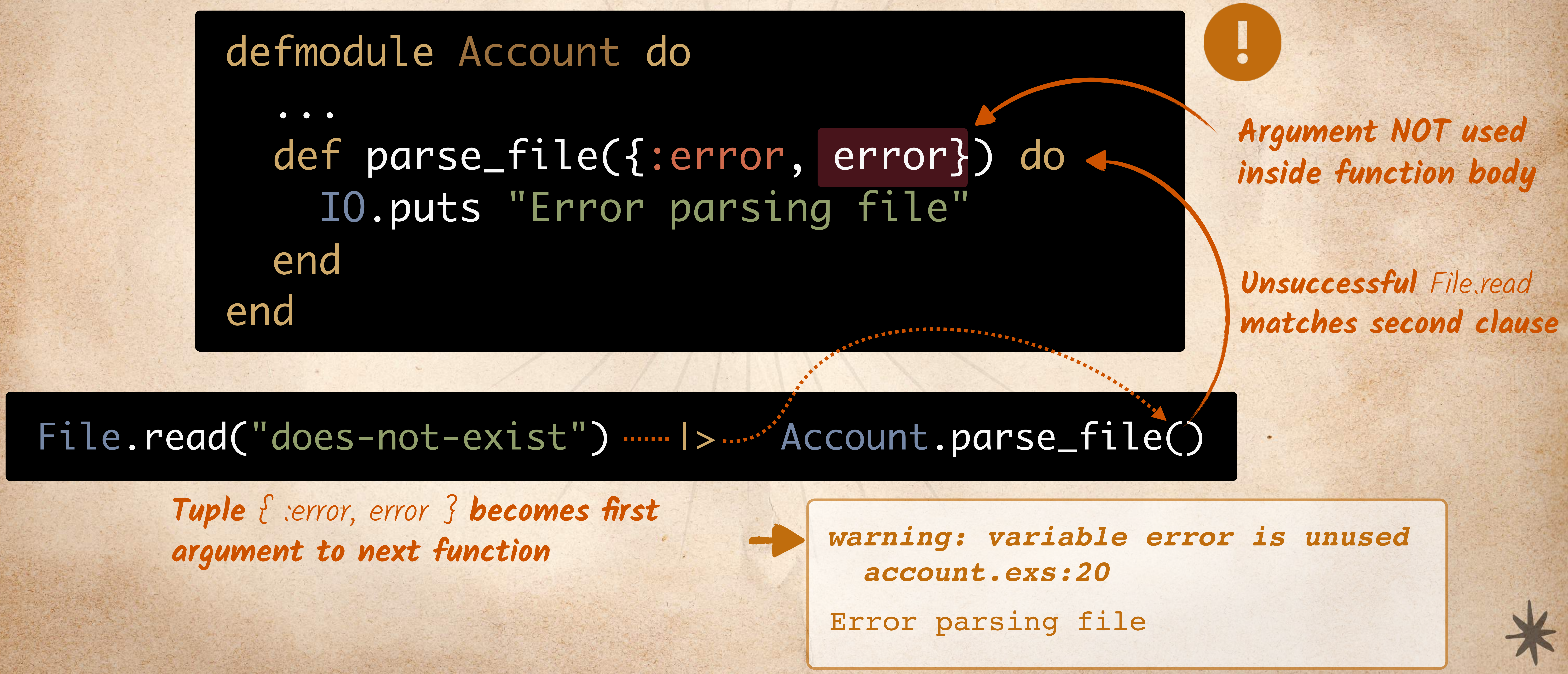
```
File.read("transactions.csv") |> Account.parse_file()
```

Tuple { :ok, content } becomes first argument to next function

```
Content: 01/12/2016,deposit,1000.00
01/12/2016,withdrawal,10.00
01/13/2016,withdrawal,25.00,
...
```

Matching Unsuccessful Return Value

Reading a file that does not exist matches the second clause. However, in this example, a warning is raised because the error variable is **not being used** from within the function.



```
defmodule Account do
  ...
  def parse_file({:error, error}) do
    IO.puts "Error parsing file"
  end
end
```

Argument NOT used inside function body

Unsuccessful File.read matches second clause

```
File.read("does-not-exist") |> Account.parse_file()
```

Tuple { :error, error } becomes first argument to next function

```
warning: variable error is unused
  account.exs:20

Error parsing file
```

Matching Unsuccessful Return Value

The underscore character is used to explicitly **ignore unused values** and avoid compiler warnings.

```elixir
defmodule Account do
  ...
  def parse_file({:error, _ }) do
    IO.puts "Error parsing file"
  end
end
```

```elixir
File.read("does-not-exist") |> Account.parse_file()
```

Explicitly ignore the value matched...

```
Error parsing file
```

...and no compiler warnings! 👍

Level 3–2

Tuples & Maps

Keyword Lists & Defaults

Listing Account Balance

An existing Account.balance function prints a balance based on a list of transactions.

```
Account.balance(transactions)
```

```
Balance: 200
```

We want to pass formatting options, like currency (dollars, euros, GBP) and symbols ($, £, €)...

```
Account.balance(transactions,                      )
```

Options argument

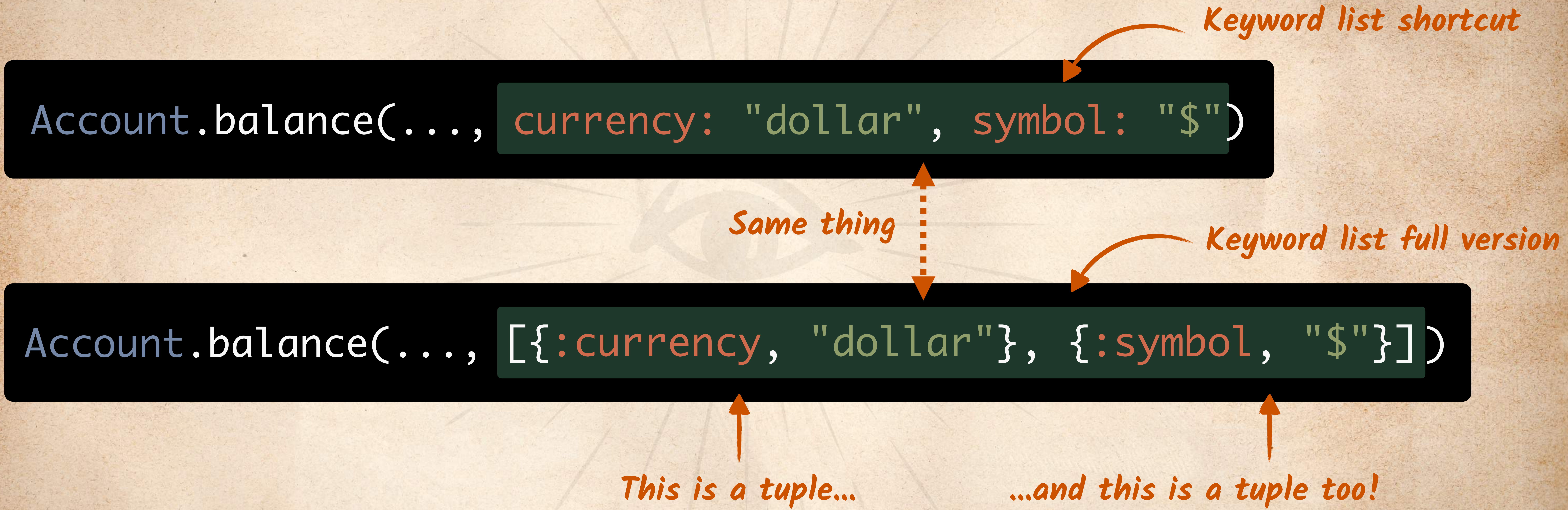
```
Balance in dollars: $200
```

```
Balance in GBP: £200
```

```
Balance in euros: €200
```

Passing Options With Keyword Lists

A keyword list is a **list of two-value tuples.** They are typically used as the last argument in function signatures, representing **options** passed to the function.



Keyword list shortcut

```
Account.balance(..., currency: "dollar", symbol: "$")
```

Same thing

Keyword list full version

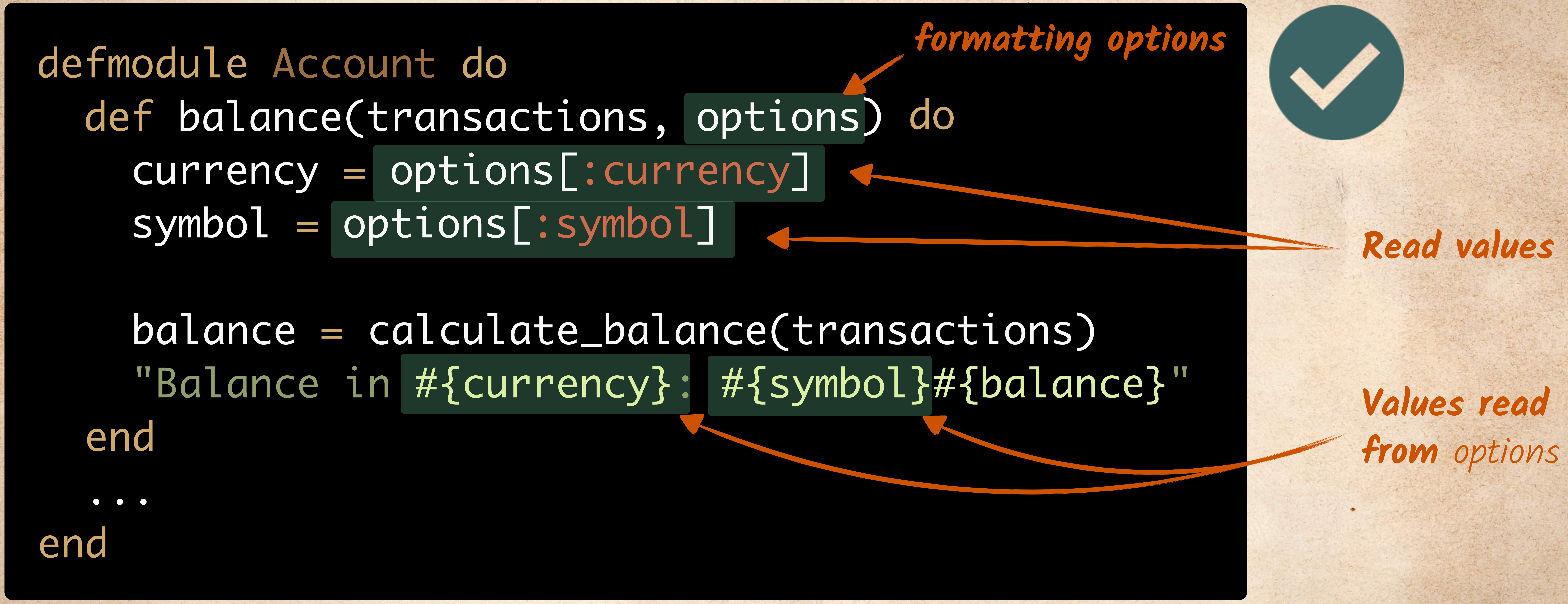
```
Account.balance(..., [{:currency, "dollar"}, {:symbol, "$"}])
```

This is a tuple...

...and this is a tuple too!

Reading Keyword Lists

To read values from keyword lists, we can use [] and the variableName[keyName] notation.



```
defmodule Account do
  def balance(transactions, options) do
    currency = options[:currency]
    symbol = options[:symbol]

    balance = calculate_balance(transactions)
    "Balance in #{currency}: #{symbol}#{balance}"
  end
  ...
end
```

formatting options

Read values

Values read from options

Running With Options

The Account.balance **function now accepts formatting options!**

```
defmodule Account do
  def balance(transactions, options) do
    currency = options[:currency]
    symbol = options[:symbol]

    balance = calculate_balance(transactions)
    "Balance in #{currency}: #{symbol}#{balance}"
  end
  ...
end
```

```
Account.balance(transactions,
         currency: "euros", symbol: "€")
```

```
Balance in euros: €200
```

Must Pass All Arguments

The code currently expects options to **always be passed.** Otherwise, it raises an error.

```
defmodule Account do
  def balance(transactions, options) do
    currency = options[:currency]
    symbol = options[:symbol]
    ...
  end
  ...
end
```

Expects second argument to always be passed

```
Account.balance(transactions)
```

Passing a single argument breaks the code

```
** (UndefinedFunctionError) function Account.balance/1
is undefined or private. Did you mean one of:

      * balance/2
```

Default Function Arguments

The `\\` symbol sets a default value to be used when none is passed during function call.

```elixir
defmodule Account do
  def balance(transactions, options \\ []) do
    currency = options[:currency]
    symbol = options[:symbol]
    ...
  end
  ...
end
```

Defaults the options argument to empty list

No values returned!

```elixir
Account.balance(transactions)
```

Code does not break anymore...

...but it's missing options!

```
Balance in          :  200
```

Defaults for Reading Keyword Lists

The logical **OR** operator `||` can be used to return a **default value** when a key is not present.

```
defmodule Account do
  def balance(transactions, options \\ []) do
    currency = options[:currency] || "dollar"
    symbol = options[:symbol] || "$"
    ...
  end
  ...
end
```

If left side of || does not return a value...

...then return this value on right side.

```
Account.balance(transactions)
```

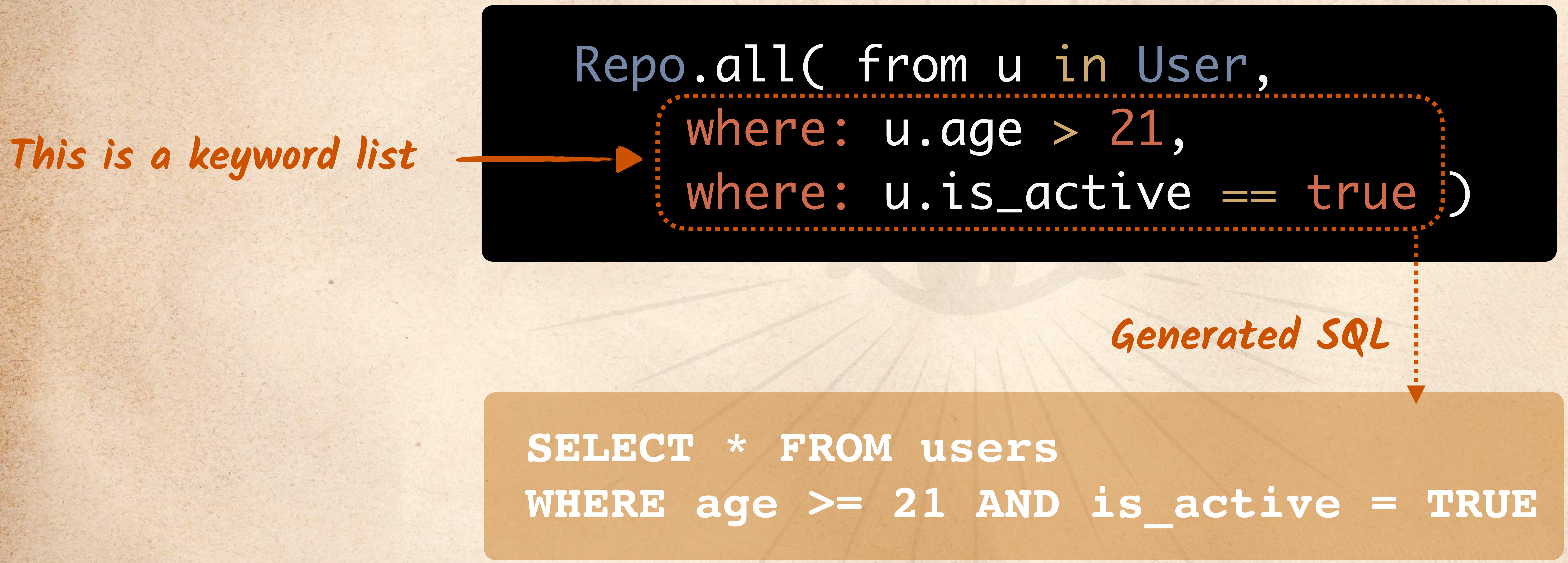
```
Balance in dollars: $200
```

animated these dotted lines and this side-text last

er defaults! 👍

Using Keyword Lists With the Ecto Library

The Ecto library uses keyword lists to build SQL statements from Elixir code.



This is a keyword list

```
Repo.all( from u in User,
   where: u.age > 21,
   where: u.is_active == true )
```

Generated SQL

```
SELECT * FROM users
WHERE age >= 21 AND is_active = TRUE
```

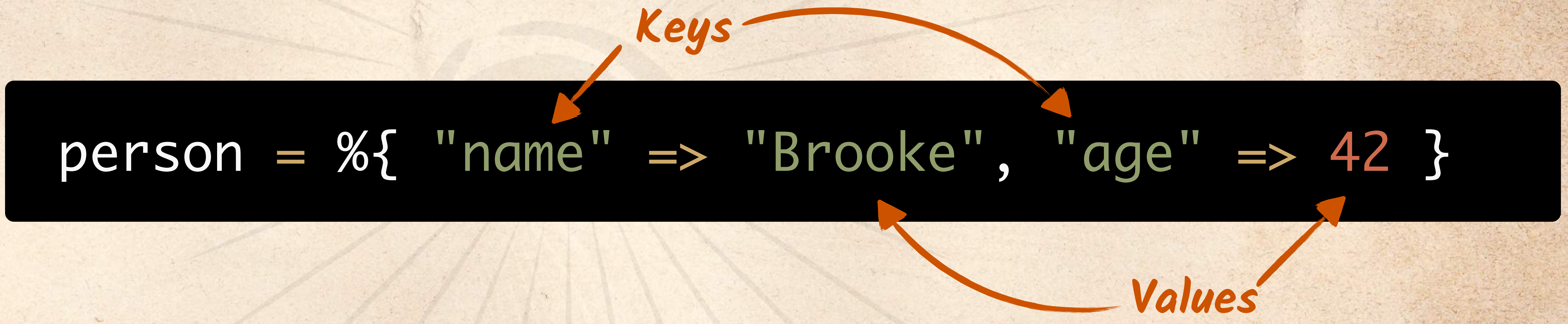
Level 3–3

Tuples & Maps

Maps

Using Maps for Structures With Named Fields

We use curly braces with the percent sign `%{}` to create maps, a collection of key-value pairs commonly used to represent a **structure with named fields.**



```
person = %{ "name" => "Brooke", "age" => 42 }
```

Reading Maps With Map.fetch and Map.fetch!

The Map module from Elixir's standard offers a set of functions for working with maps.

Map.fetch **returns a tuple** when key is present

```
Map.fetch(person, "name")
```

```
{:ok, "Brooke"}
```

...and the :error atom when it’s not.

```
Map.fetch(person, "banana")
```

```
:error
```

Map.fetch! **returns a value** when key is present

```
Map.fetch!(person, "name")
```

```
"Brooke"
```

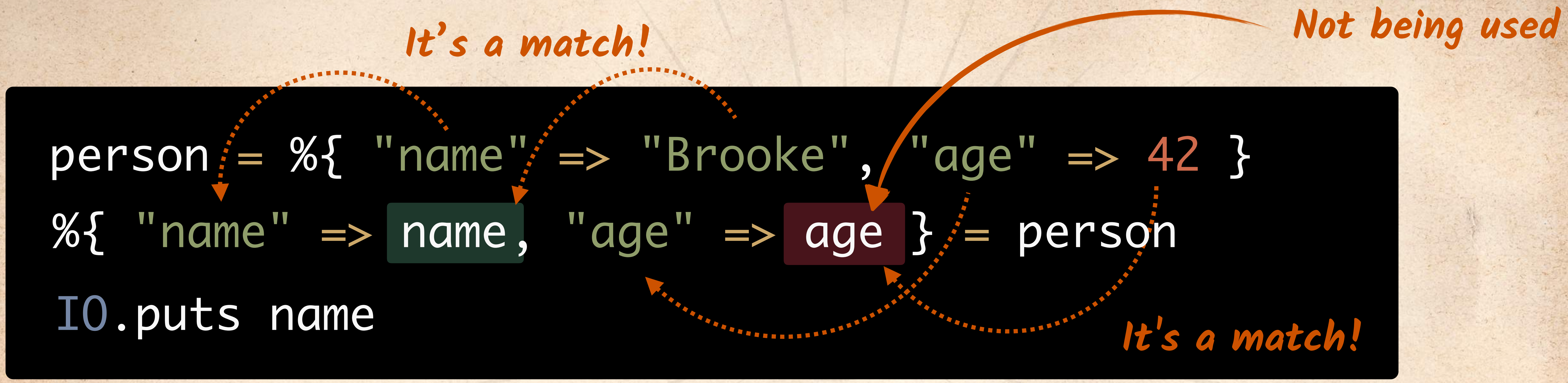
...and raises an error when it’s not.

```
Map.fetch!(person, "banana")
```

```
** (KeyError) key "banana" not found in: %{"name" => "Brooke,
      "age" => 42}
    (elixir) lib/map.ex:164: Map.fetch!/2
```

Reading Maps With Pattern Matching

We can also use **pattern matching** to read values from a map.



It's a match!

Not being used

```
person = %{ "name" => "Brooke", "age" => 42 }
%{ "name" => name, "age" => age } = person
IO.puts name
```

It's a match!

```
warning: variable age is unused

Brooke
```

Warnings will NOT stop programs from running, but it's best not to have them.

Matching Portions of a Map

Unlike tuples, with maps we can pattern match only **the portion** we are interested in.

...other keys are ignored.

```
person = %{ "name" => "Brooke", "age" => 42 }
%{ "name" => name } = person
IO.puts name
```

```
Brooke
```

✓

Only reads the value for the name key on the map...

```
person = [{:name, "Booke"}, {:age, 42}]
[{:name, name}] = person
IO.puts name
```

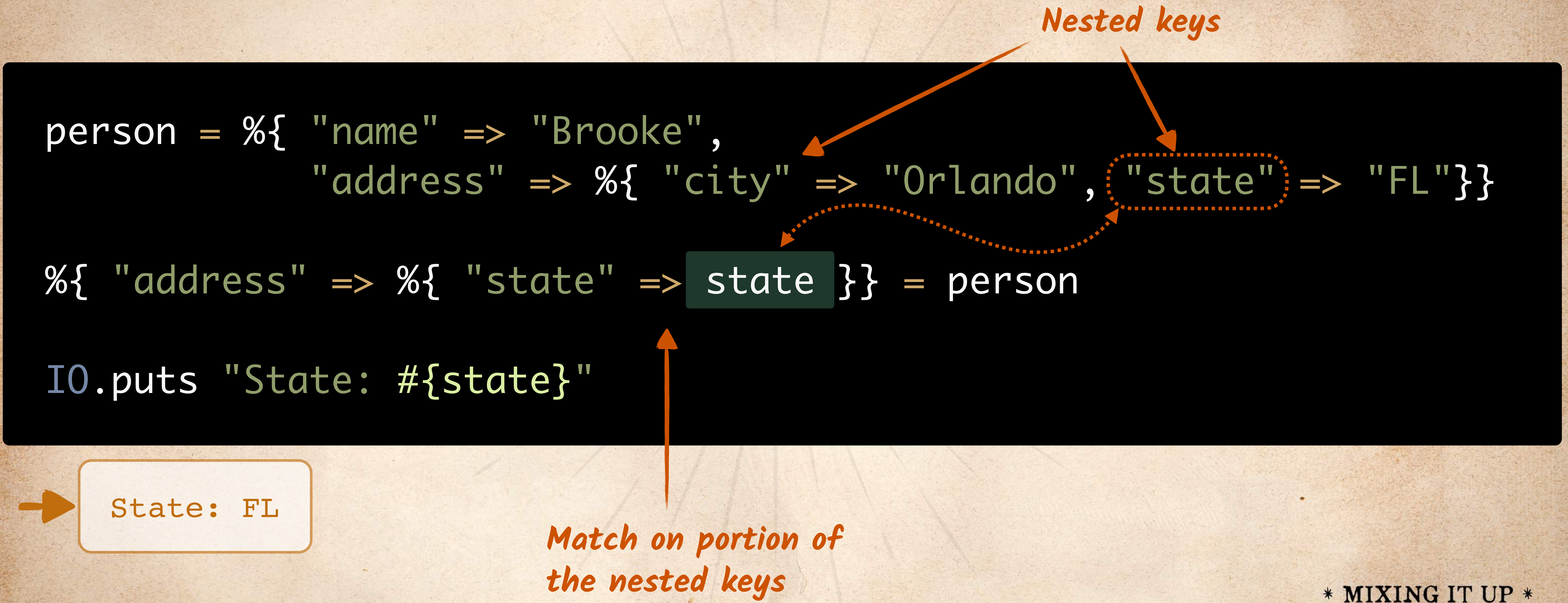
List of tuples do not support partial match

```
** (MatchError) no match of right hand
   side value: [name: "Booke", age: 42]
```

* MIXING IT UP *
with
* ELIXIR *

Advanced Pattern Matching With Maps

Even deeply nested keys in maps can be read using pattern matching.



```
person = %{ "name" => "Brooke",
            "address" => %{ "city" => "Orlando", "state" => "FL"}}

%{ "address" => %{ "state" => state }} = person

IO.puts "State: #{state}"
```

Nested keys

Match on portion of
the nested keys

```
State: FL
```

Keyword Lists or Maps?

Here’s a quick summary to help pick the appropriate data type.

```
Account.balance(transactions,
  currency: "dollar", symbol: "$")
```

When to use keyword lists?

To pass optional values to functions.

```
person = %{ "name" => "Brooke", "age" => 42 }
%{ "name" => name } = person
```

When to use maps?

To represent a structure as a key-value storage.

Level 4–1

Control Flow

The case Statement

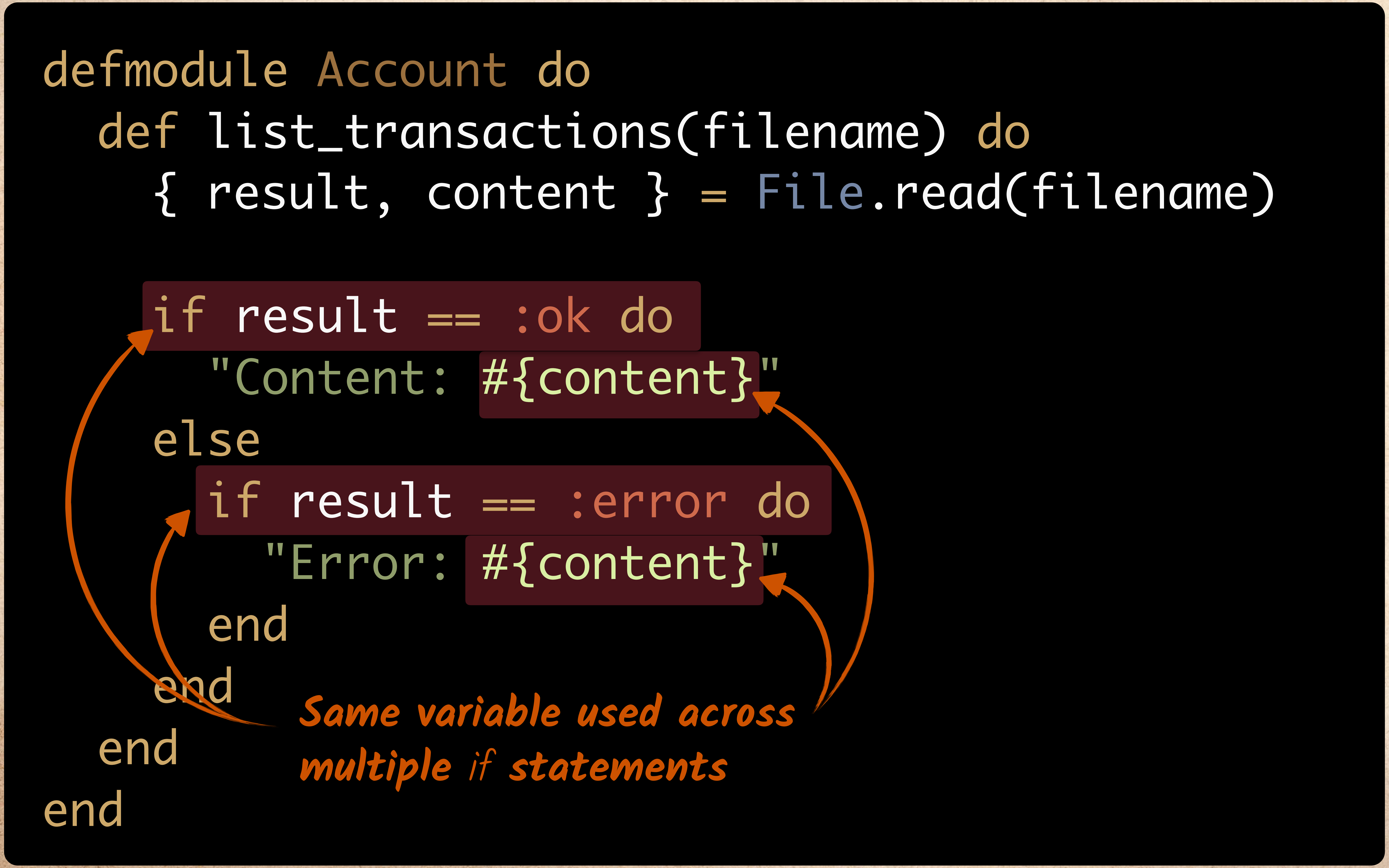
Listing Content From a File

The function Account.list_transactions() **takes a file name as argument and lists its contents.**

```
defmodule Account do
  def list_transactions(filename) do
    { result, content } = File.read(filename)

    if result == :ok do
      "Content: #{content}"
    else
      if result == :error do
        "Error: #{content}"
      end
    end
  end
end
```

Nested if Statements Are Hard to Read

Repeating variables (result, content) in nested if statements illustrate a common code smell.

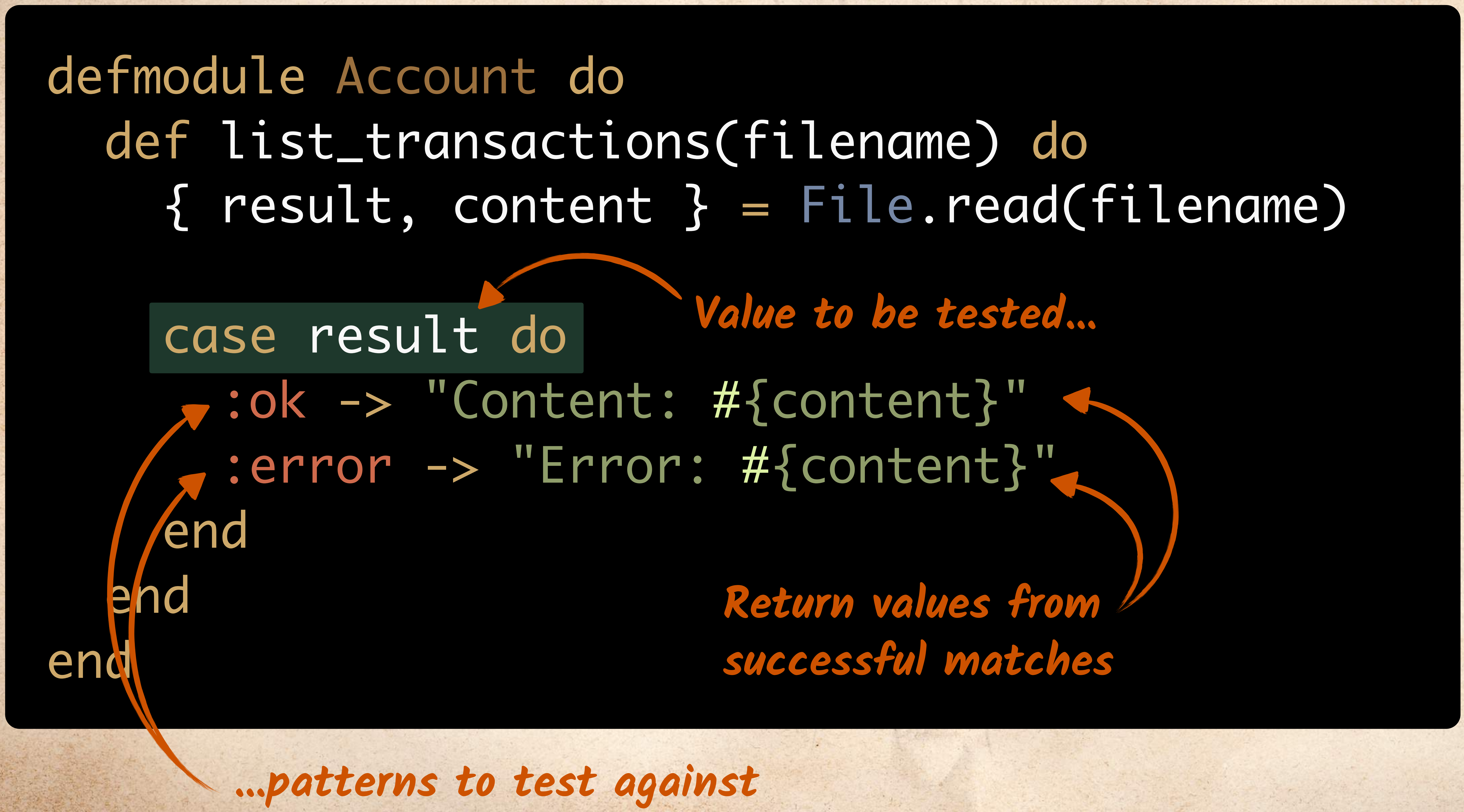


```
defmodule Account do
  def list_transactions(filename) do
    { result, content } = File.read(filename)

    if result == :ok do
      "Content: #{content}"
    else
      if result == :error do
        "Error: #{content}"
      end
    end
  end
end
```

Same variable used across multiple if statements

Using case to Test Values Against Patterns

The case statement tests a **value** against a set of **patterns**.



```
defmodule Account do
  def list_transactions(filename) do
    { result, content } = File.read(filename)

    case result do
      :ok -> "Content: #{content}"
      :error -> "Error: #{content}"
    end
  end
end
```

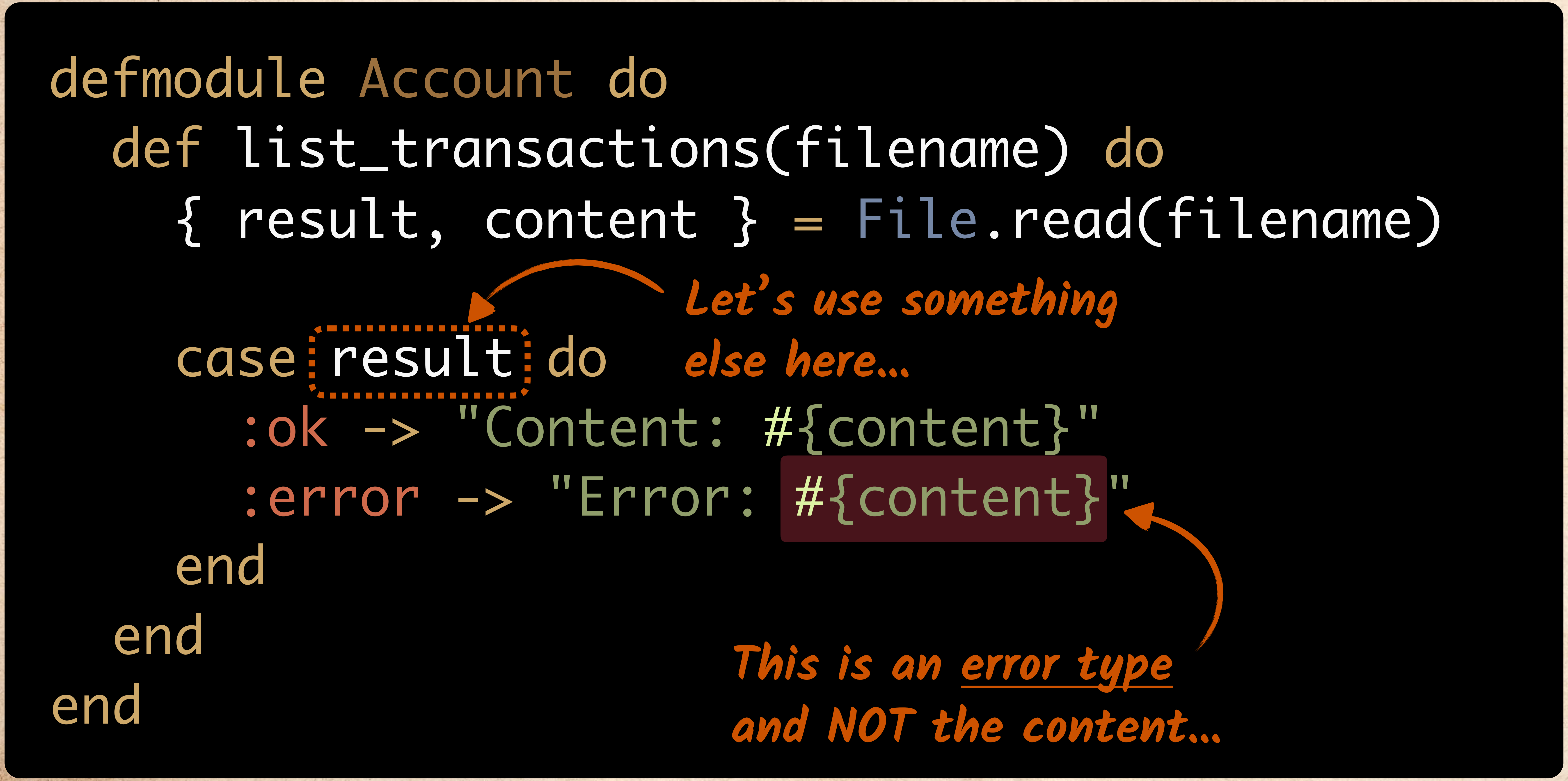
Value to be tested...

Return values from successful matches

...patterns to test against

Misleading Variable Names

Using result **as the test value for the** case **statement is leading to the use of the same variable name (**content**) for the content of the file (when** result **is** :ok**) or for the error (when** result **is** :error**).**



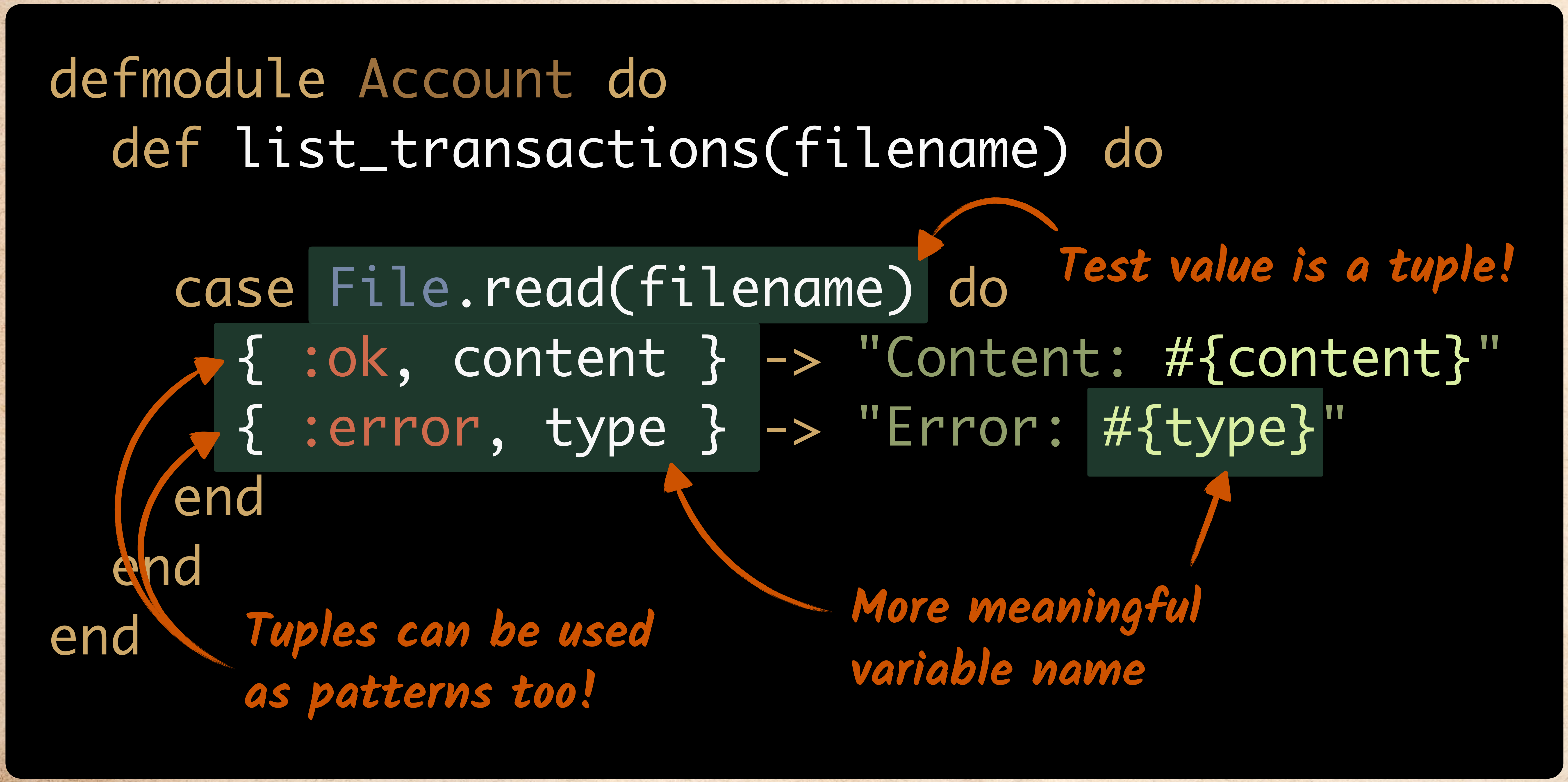
```elixir
defmodule Account do
  def list_transactions(filename) do
    { result, content } = File.read(filename)

    case result do
      :ok -> "Content: #{content}"
      :error -> "Error: #{content}"
    end
  end
end
```

Let's use something else here...

This is an error type and NOT the content...

Better Variable Names With case

The case statement accepts tuples for the test values as well as for the patterns to be tested against. This gives us **more flexibility for naming variables**.



```
defmodule Account do
  def list_transactions(filename) do

    case File.read(filename) do
      { :ok, content } -> "Content: #{content}"
      { :error, type } -> "Error: #{type}"
    end
  end
end
```

Test value is a tuple!

Tuples can be used as patterns too!

More meaningful variable name

No Code Smell & Works as Expected

```
defmodule Account do
  def list_transactions(filename) do
    case File.read(filename) do
      { :ok, content } -> "Content: #{content}"
      { :error, type } -> "Error: #{type}"
    end
  end
end
```

```
Account.list_transactions("transactions.csv")
```

```
Content: 01/12/2016,deposit,1000.00
01/12/2016,withdrawal,10.00
01/13/2016,withdrawal,25.00,
...
```

```
Account.list_transactions("does-not-exist")
```

```
Error: enoent
```

Using case with Guard Clauses

The case statement allows extra conditions to be specified with a **guard clause**.

```
defmodule Account do
  def list_transactions(filename) do
    case File.read(filename) do
      { :ok, content }
       when byte_size(content) > 10 -> "Content: (...)"
      { :ok, content } -> "Content: #{content}"
      { :error, type } -> "Error: #{type}"
    end
  end
end
```

built-in function

does not list transactions

returns true when file content is greater than 10 characters.

```
Account.list_transactions("loooong-list.csv")
```

```
Content: (...)
```

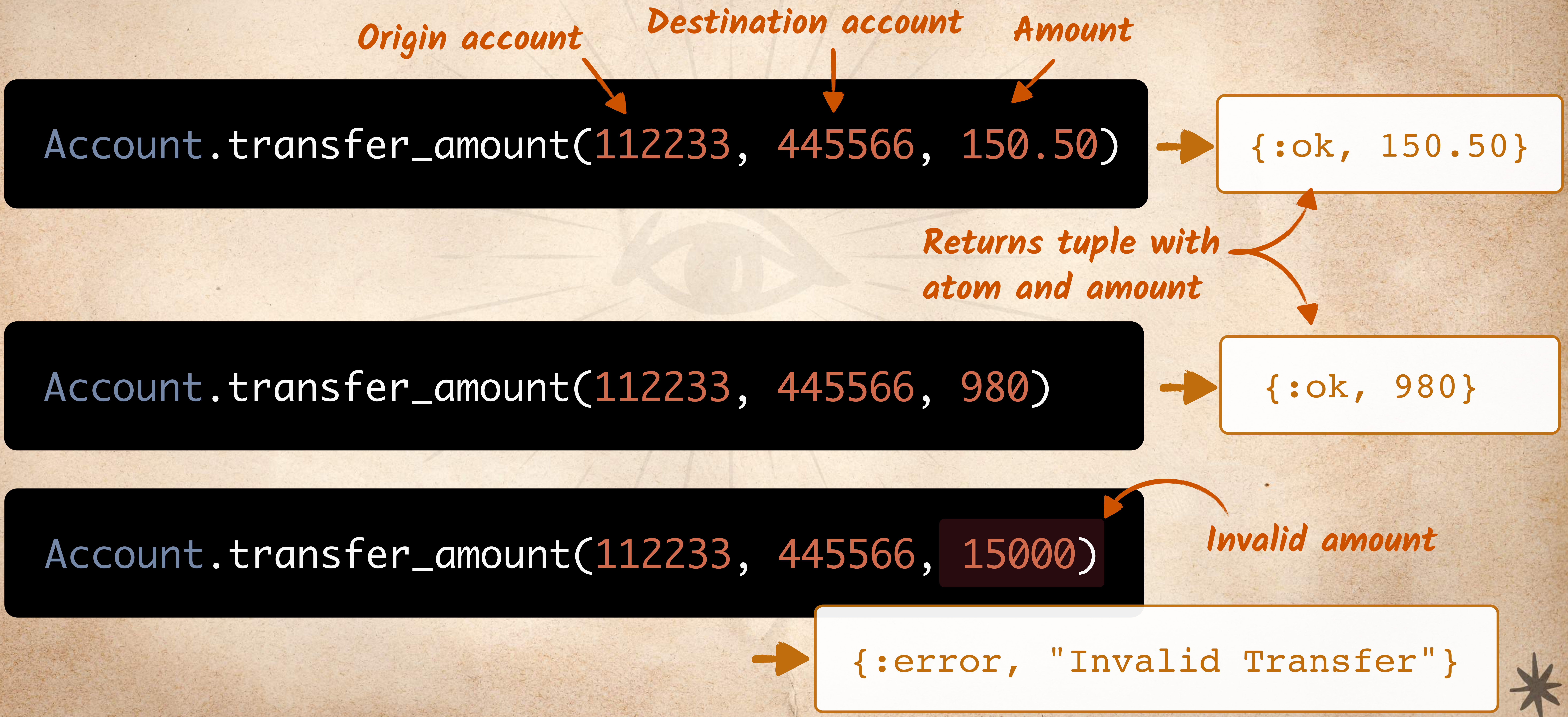
Level 4–2

Control Flow

The cond Statement

Transferring Between Accounts

We'll write a function to transfer money between accounts.



Origin account

Destination account

Amount

```
Account.transfer_amount(112233, 445566, 150.50)
```

```
{:ok, 150.50}
```

Returns tuple with atom and amount

```
Account.transfer_amount(112233, 445566, 980)
```

```
{:ok, 980}
```

```
Account.transfer_amount(112233, 445566, 15000)
```

Invalid amount

```
{:error, "Invalid Transfer"}
```

Transfer Depends on Validation

The **validation** for a transfer involves the amount transferred and the hour of the day.

```elixir
defmodule Account do
  def transfer_amount(from_account, to_account, amount) do
    hourOfDay = DateTime.utc_now.hour

    if !valid_transfer?(amount, hourOfDay) do
      {:error, "Invalid Transfer"}
    else
      perform_transfer(from_account, to_account, amount)
    end
  end
  ...
end
```

Part of Elixir's standard library

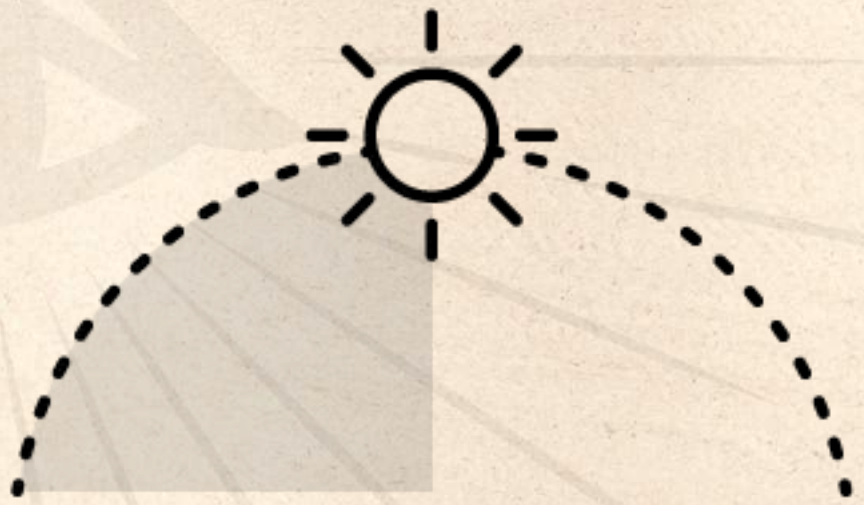
Defined elsewhere in this module

The Logic for the valid_transfer? **Function**

The amount allowed to be transferred depends on the time of the day.

Morning *(before noon)*
No more than $5000

Afternoon *(before 6pm)*
No more than $1000



Evening *(after 6pm)*
No more than $300

And the Nested if Statements Attack Again!

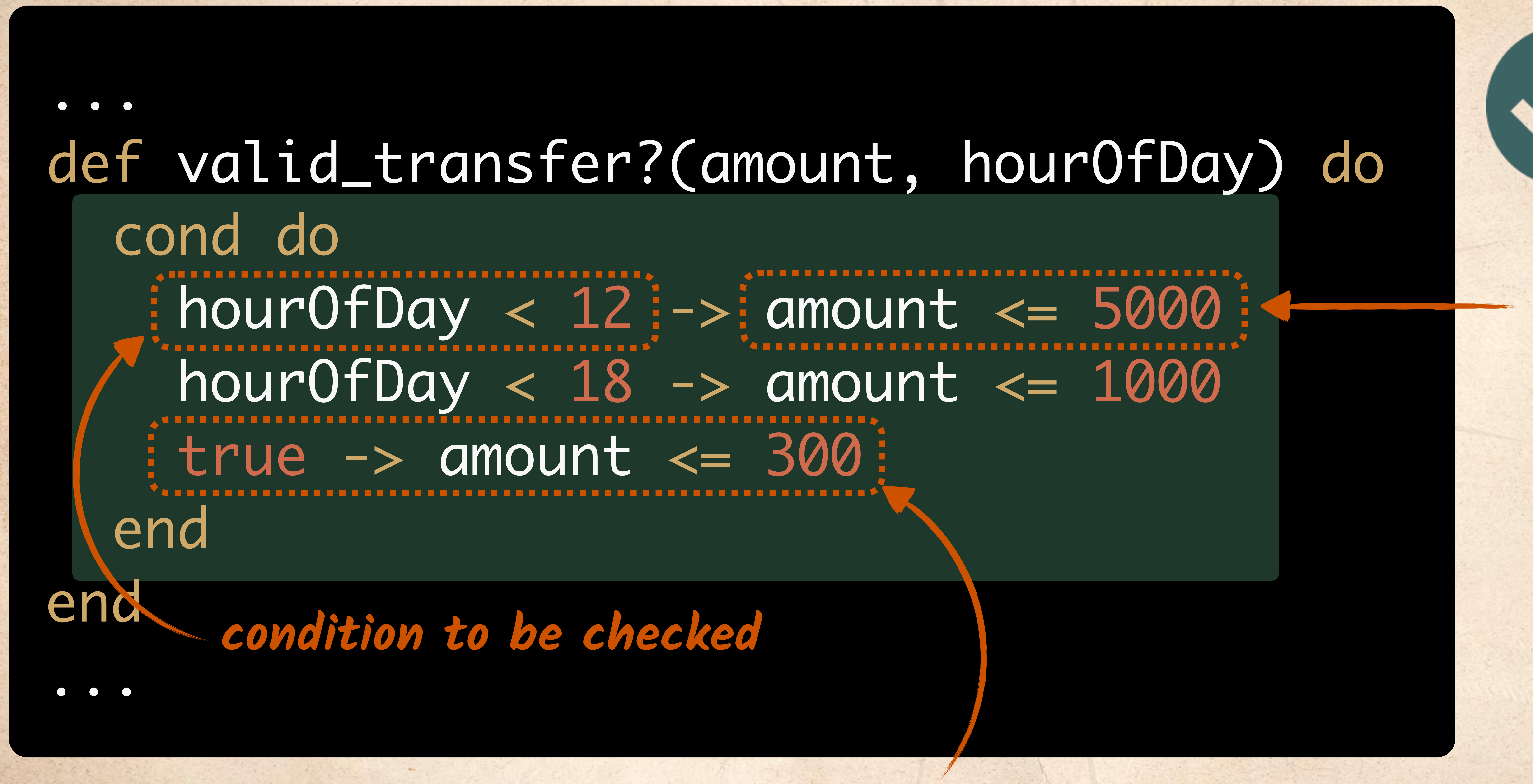
We could implement this using nested if statements… but we've been there before, remember?

```
...
def valid_transfer?(amount, hourOfDay) do
  if hourOfDay < 12 do
    amount <= 5000
  else
    if hourOfDay < 18 do
      amount <= 1000
    else
      amount <= 300
    end
  end
end
...
```

Valid code, but hard to read and maintain!

The cond Statement

The cond statement checks multiple **conditions** and finds **the first one** that evaluates to *true*.



```
...
def valid_transfer?(amount, hourOfDay) do
  cond do
    hourOfDay < 12 -> amount <= 5000
    hourOfDay < 18 -> amount <= 1000
    true -> amount <= 300
  end
end
...
```

condition to be checked

Block runs when condition is *true*

Catch all when none of the previous conditions are *true*

* MIXING IT UP *
with
* ELIXIR *

Running the Transfer

The Account.transfer_amount **function is now complete!**

```
Account.transfer_amount(112233, 445566, 150.50)
```

```
{:ok, 150.50}
```

```
Account.transfer_amount(112233, 445566, 980)
```

```
{:ok, 980}
```

Can't transfer this much after 12pm

```
Account.transfer_amount(112233, 445566, 1500)
```

```
{:error, "Invalid Transfer"}
```

To case **or to** cond?

We use case for **matching** on multiple **patterns:**

```
case File.read(filename) do
  { :ok, content } -> "Content: #{content}"
  { :error, type } -> "Error: #{type}"
end
```

We use cond for **checking** multiple **conditions:**

```
cond do
  hourOfDay < 12 -> amount <= 5000
  hourOfDay < 18 -> amount <= 1000
  true -> amount <= 300
end
```

Level 5–1

The Mix Tool

Running Tasks & Organizing Projects

Benefits of a Well-structured Project

Keeping a well-organized project and adopting a standard for project organization can help in many ways. Here are three major benefits:

- Easier to navigate project files.
- Facilitates collaboration from other developers on the team.
- Facilitates onboarding new members.

Using Mix to Create a New Project

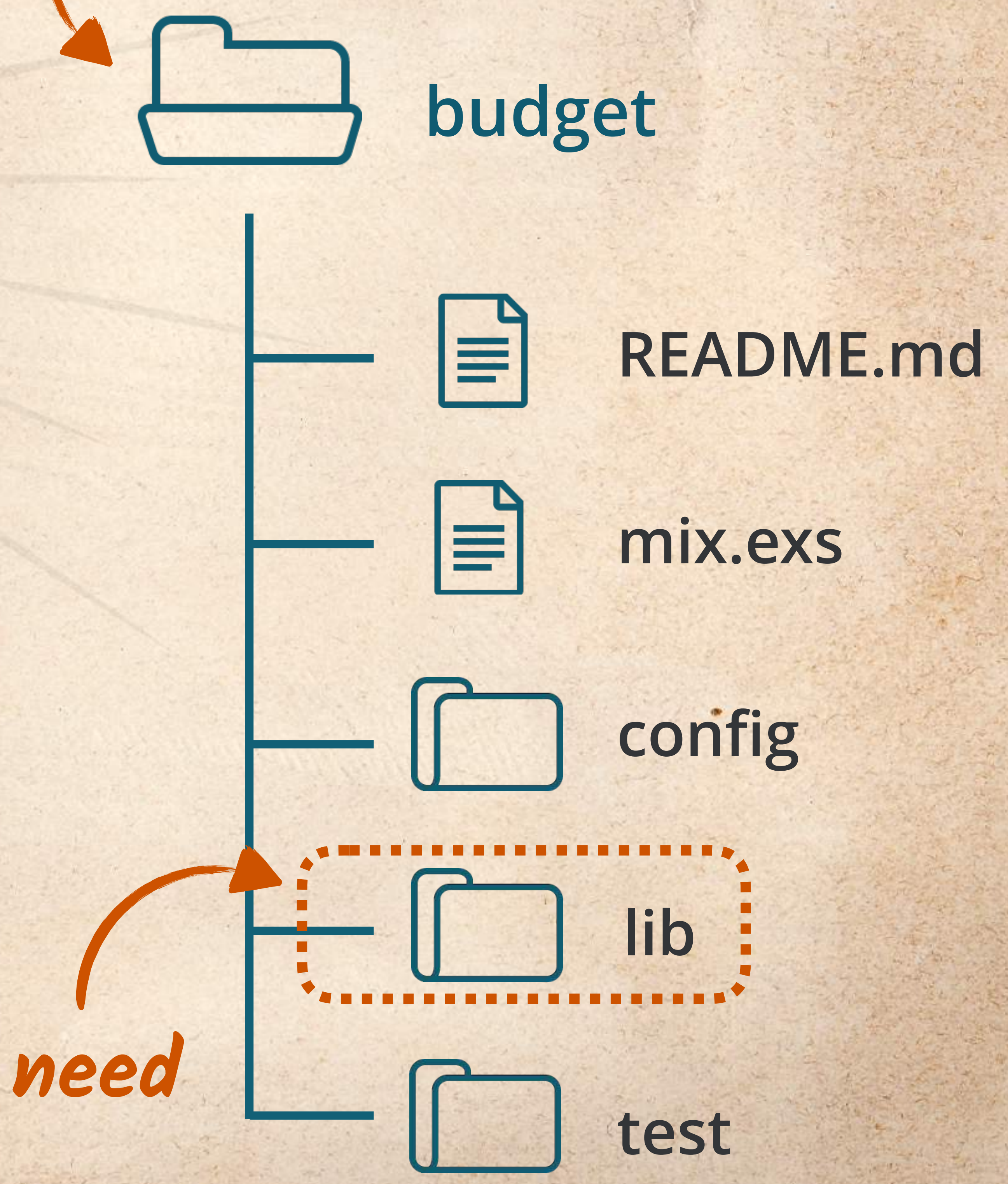
Mix is a **build tool** installed with Elixir that provides tasks for creating, compiling, and testing Elixir projects, managing its dependencies, and more.

Name of the project

```
$ mix new budget
```

```
* creating README.md
...

Your Mix project was created successfully.
You can use "mix" to compile it, test it, and more:

    cd budget
    mix test

Run "mix help" for more commands.
```

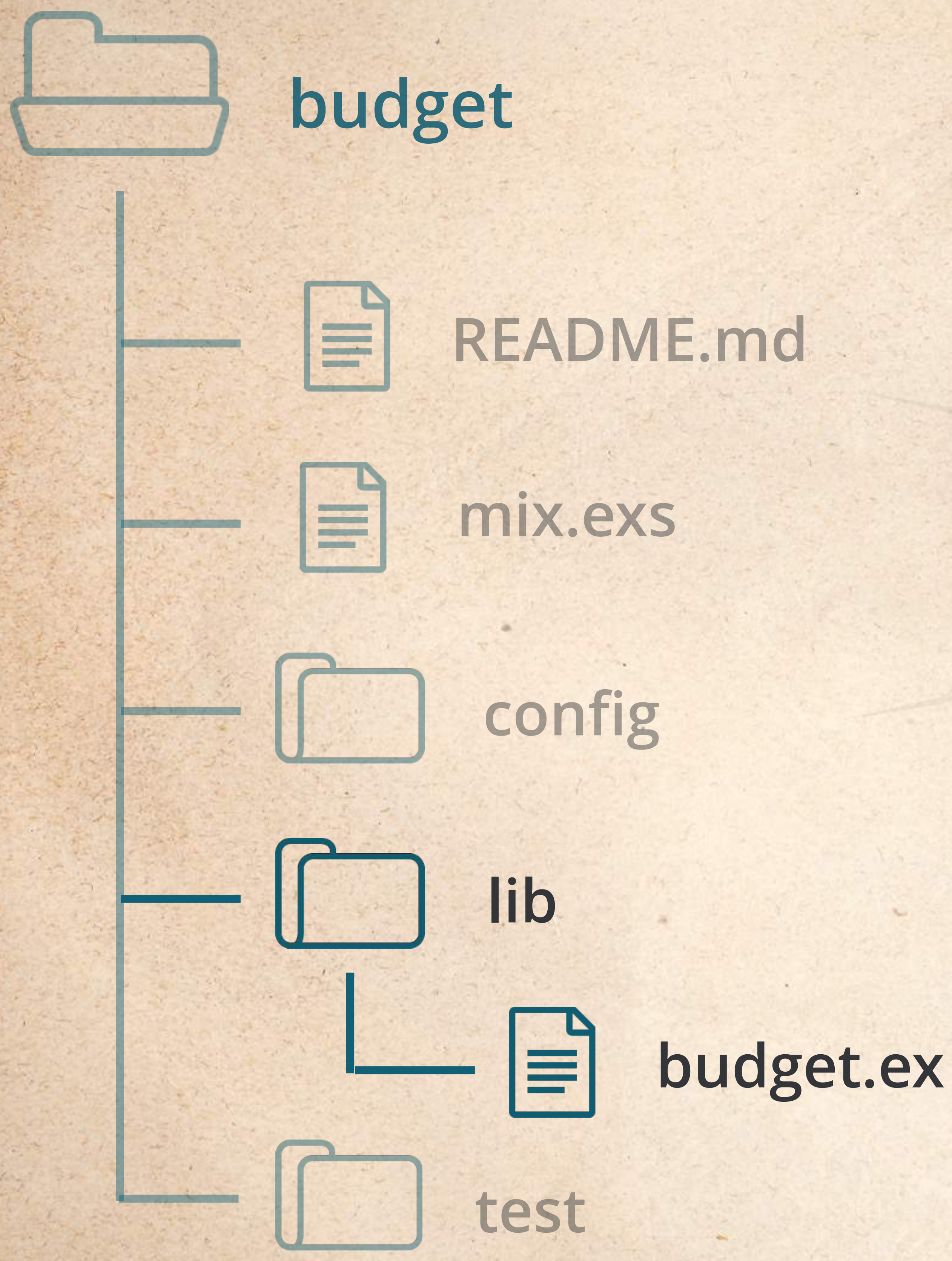
Directories and files created for us!



The only folder we need to access for now

Writing a New Function

We'll define current_balance **as part of the** Budget **module, created for us by the** mix new **command.**



Created by Mix

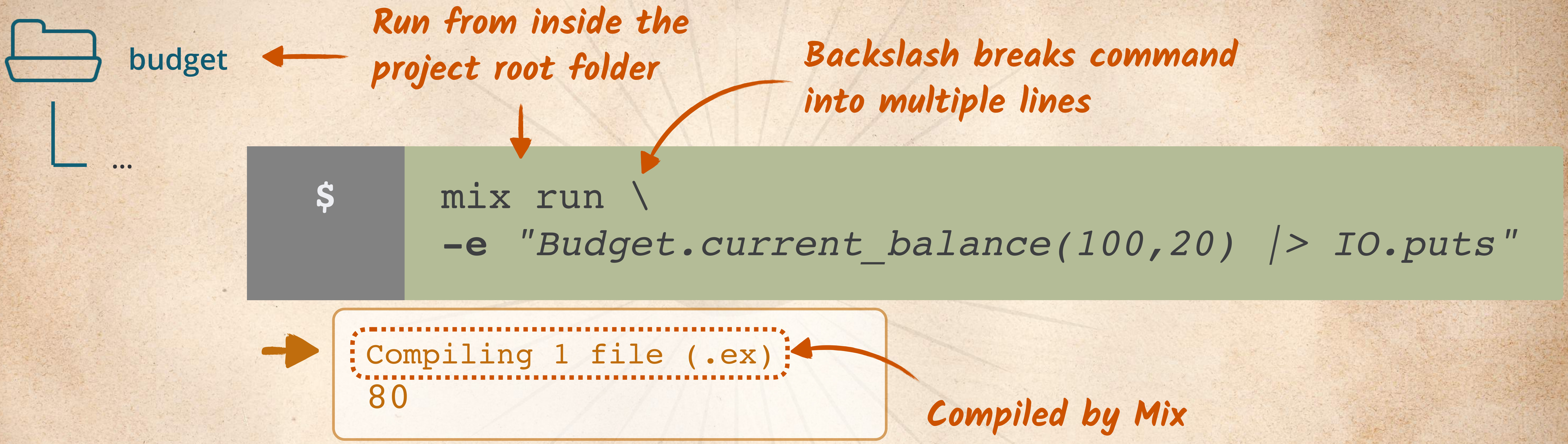
lib/budget.ex

```elixir
defmodule Budget do
  def current_balance(initial, spending) do
    initial - spending
  end
end
```

New function defined by us

Running Programs With mix run

The -e option tells the mix run command to evaluate a given code in the context of the application.


budget
...
Run from inside the project root folder
Backslash breaks command into multiple lines
Compiled by Mix

```
$ mix run \
  -e "Budget.current_balance(100,20) |> IO.puts"
```

```
Compiling 1 file (.ex)
80
```

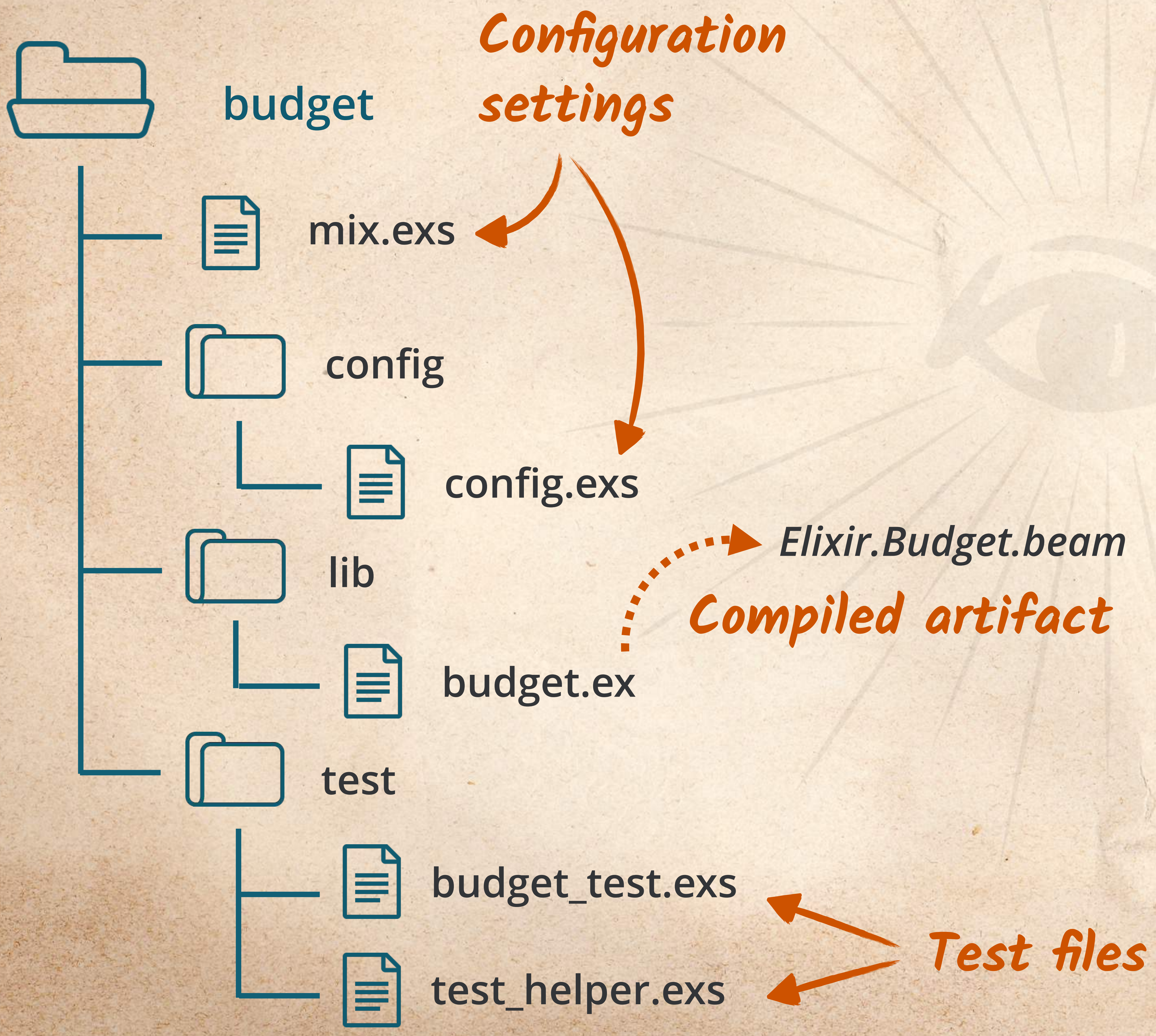
What the mix run command does:

1. Compiles the budget application.
2. Loads the generated bytecode into the Erlang Virtual Machine.
3. Detects the -e option and evaluates the argument as code.

* MIXING IT UP * with * ELIXIR *

The Difference Between File Extensions

Both .ex and .exs file extensions are treated **the same way**. The difference is intention: .ex files are meant to be **compiled** while .exs files are used for **scripting.**



.ex files

- Generates production artifacts (.*beam* files)
- Examples: lib files

.exs files

- Does NOT generate production artifacts
- Examples: configuration files, test files

Mix Help!

We can run the **mix help** command to see the list of all available tasks.

```
$ mix help
```

```
mix                   # Runs the default task (current: "mix run")
mix app.start         # Starts all registered apps
mix app.tree          # Prints the application tree
mix archive           # Lists installed archives
mix archive.build     # Archives this project into a .ez file
mix archive.install   # Installs an archive locally
mix archive.uninstall # Uninstalls archives

...
```

Level 5–2

The Mix Tool

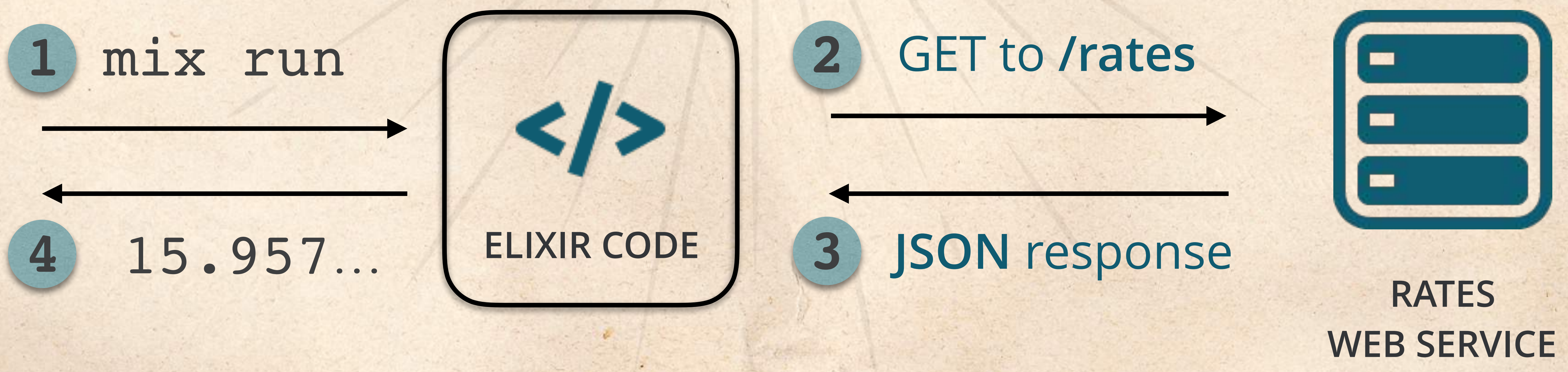
Working With Third-party Dependencies

Converting From Euro to Dollar

Let’s write a new function from_euro_to_dollar() that takes an amount in € euros as its single argument and converts it to US$ dollars. We’ll fetch the rate of the day from an **external web service API.**

```
$ mix run -e "Budget.Conversion.from_euro_to_dollar(15) |> IO.puts"
```

```
15.957446808510639
```



1 mix run

4 15.957…

ELIXIR CODE

2 GET to /rates

3 JSON response

RATES
WEB SERVICE

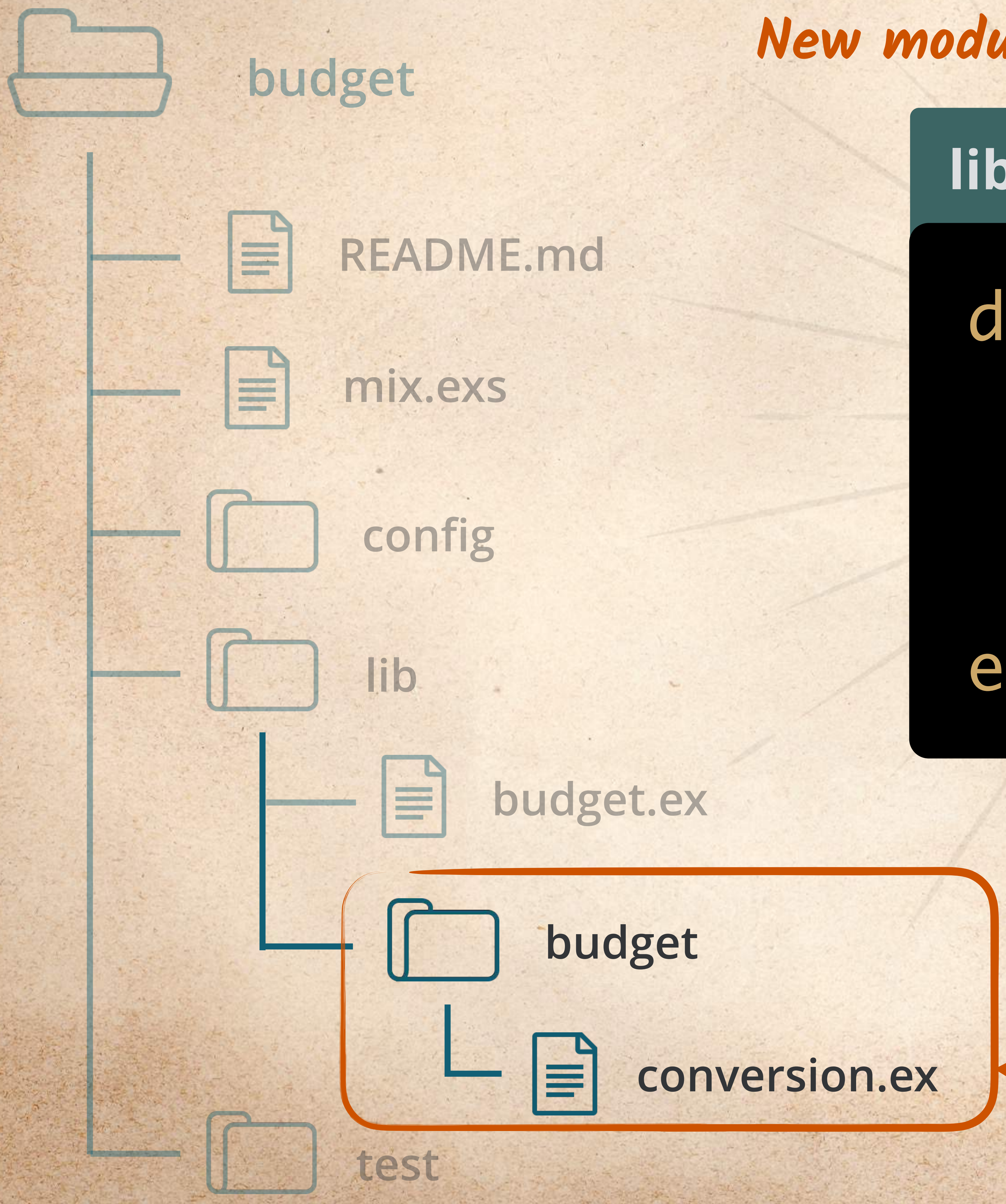
Creating a New Module

The new function will be part of the Conversion **module, which itself is a submodule of** Budget.

New module part of the *Budget* ***module***

lib/budget/conversion.ex

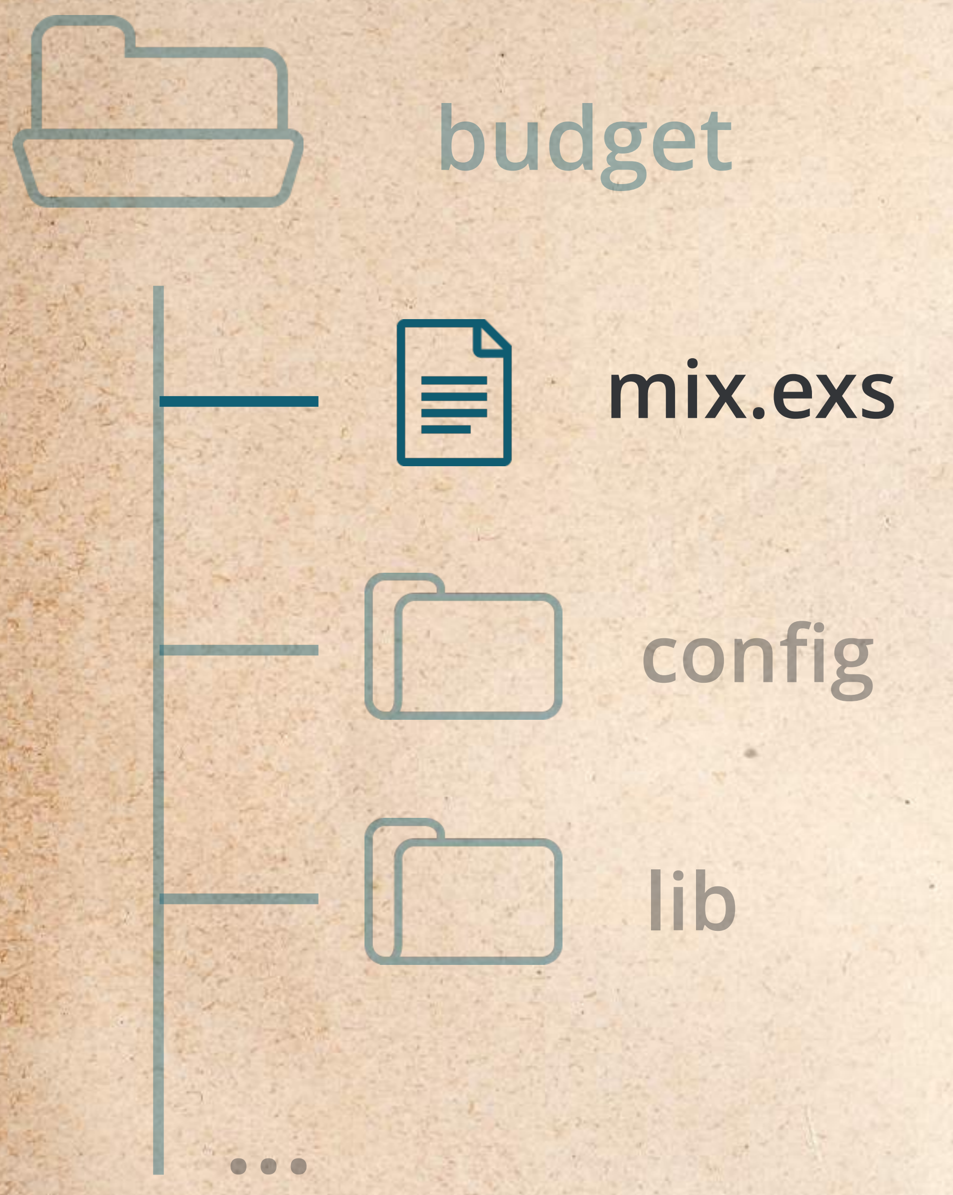
```
defmodule Budget.Conversion do
  def from_euro_to_dollar(amount) do
    ...
  end
end
```

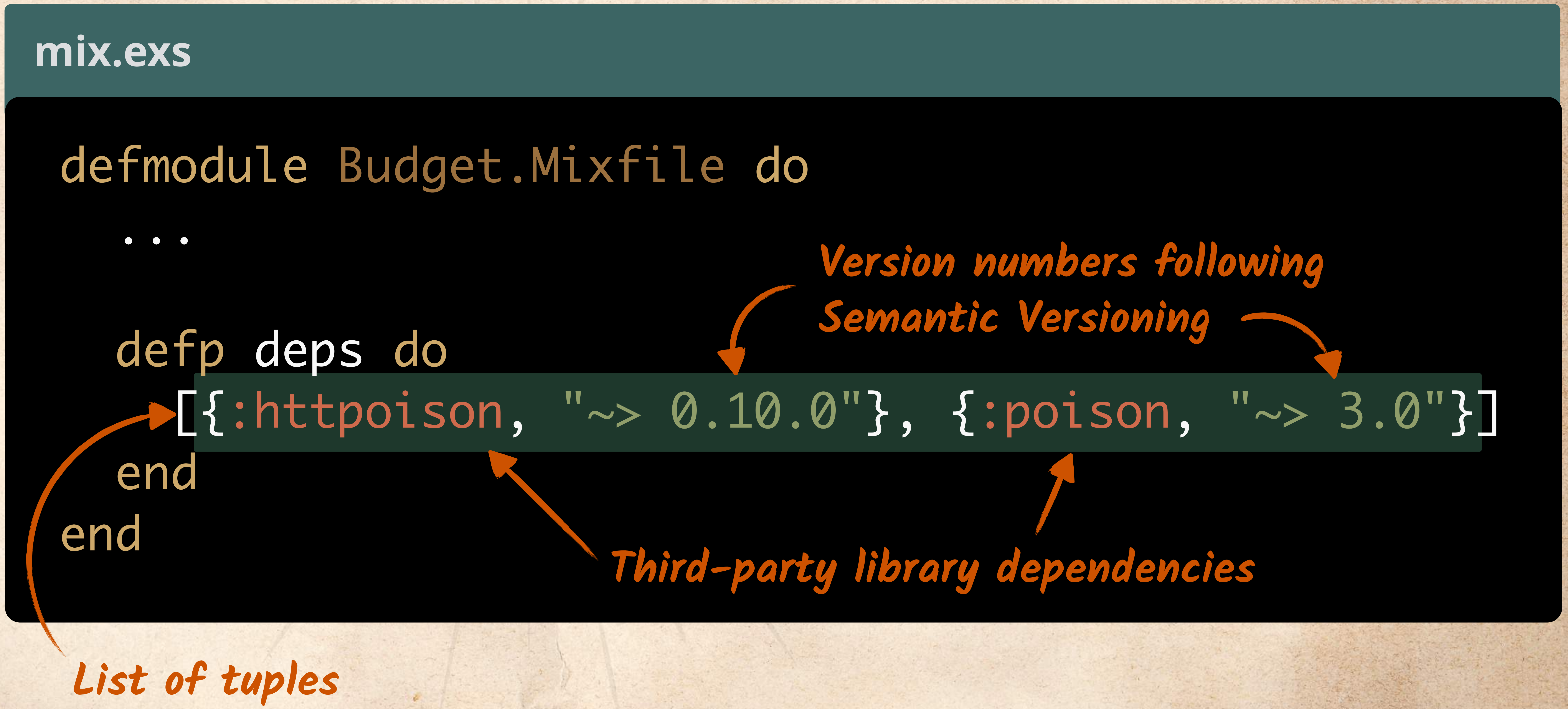


Create new folder and new file

Declaring Third-party Dependencies

We use the mix.exs file to declare **library dependencies** our program depends on.




mix.exs

```
defmodule Budget.Mixfile do
  ...

  defp deps do
    [{:httpoison, "~> 0.10.0"}, {:poison, "~> 3.0"}]
  end
end
```

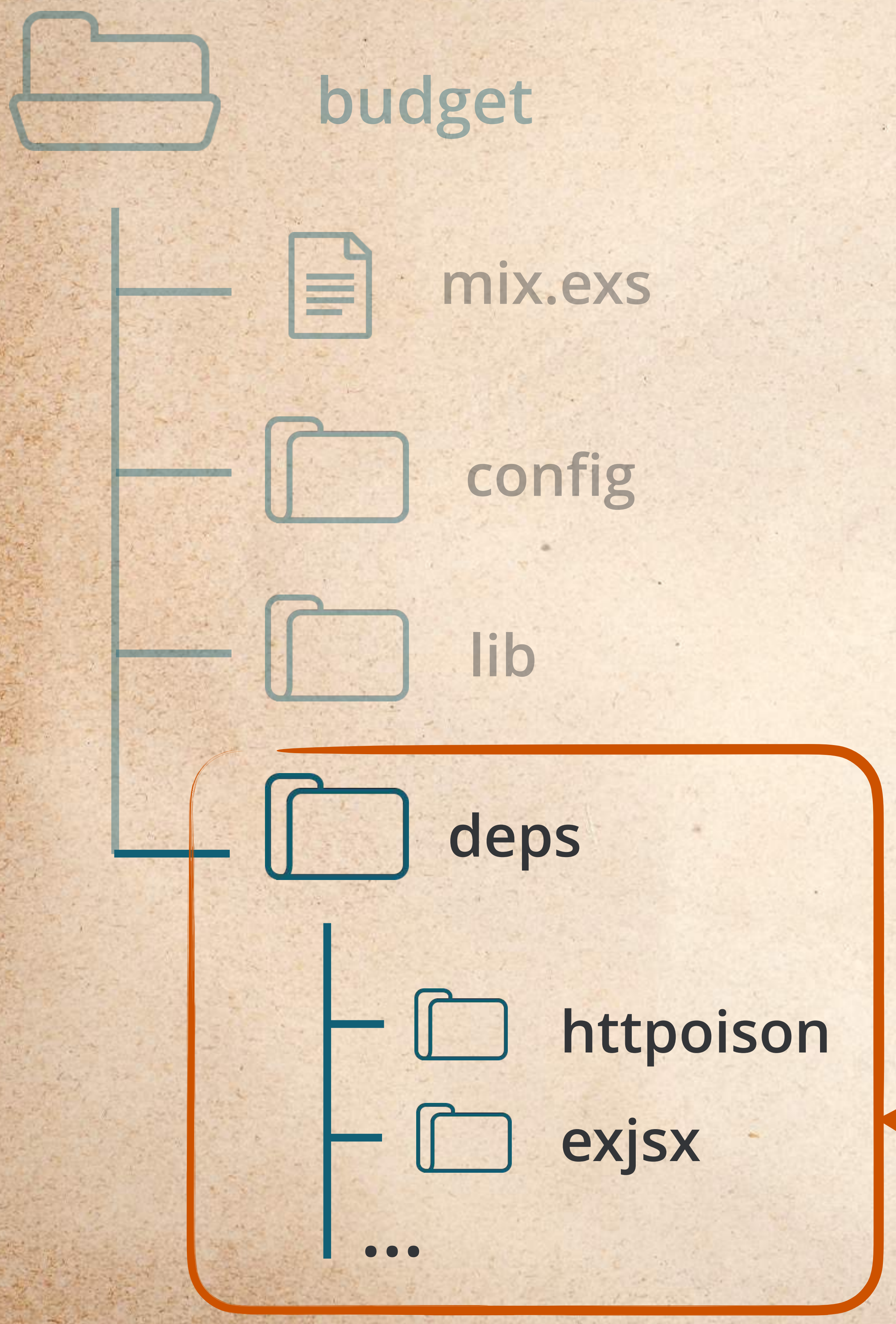
Version numbers following Semantic Versioning

Third-party library dependencies

List of tuples

Installing Third-party Dependencies

The command mix deps.get **fetches dependencies from a remote repository and installs them locally.**



```
$ mix deps.get
```

```
Running dependency resolution
* Getting httpoison (Hex package)
  Checking package (https://repo.hex.pm/tarballs/httpoison-0.10.0.tar)
  Using locally cached package
* Getting poison (Hex package)
  Checking package (https://repo.hex.pm/tarballs/poison-3.0.0.tar)
  Using locally cached package
...
```

Each third-party dependency is stored inside the deps *directory.*

Making HTTP Calls With the HTTPoison Library

The HTTPoison library is what we'll use to make HTTP calls to the remote web service.

lib/budget/conversion.ex

```elixir
defmodule Budget.Conversion do
  def from_euro_to_dollar(amount) do
    url = "cs-currency-rates.codeschool.com/currency-rates"
    case HTTPoison.get(url) do
      {:ok, response} -> parse(response) |> convert(amount)
      {:error, _} -> "Error fetching rates"
    end
  end
  ...
end
```

Takes result of parse(response) as first argument

Using pattern matching to determine whether the HTTP call was successful

Parsing JSON With the JSX library

We use **pattern matching** to store the response body on the json_response **variable and the** Poison **library to parse JSON to an Elixir *tuple*.**

lib/budget/conversion.ex

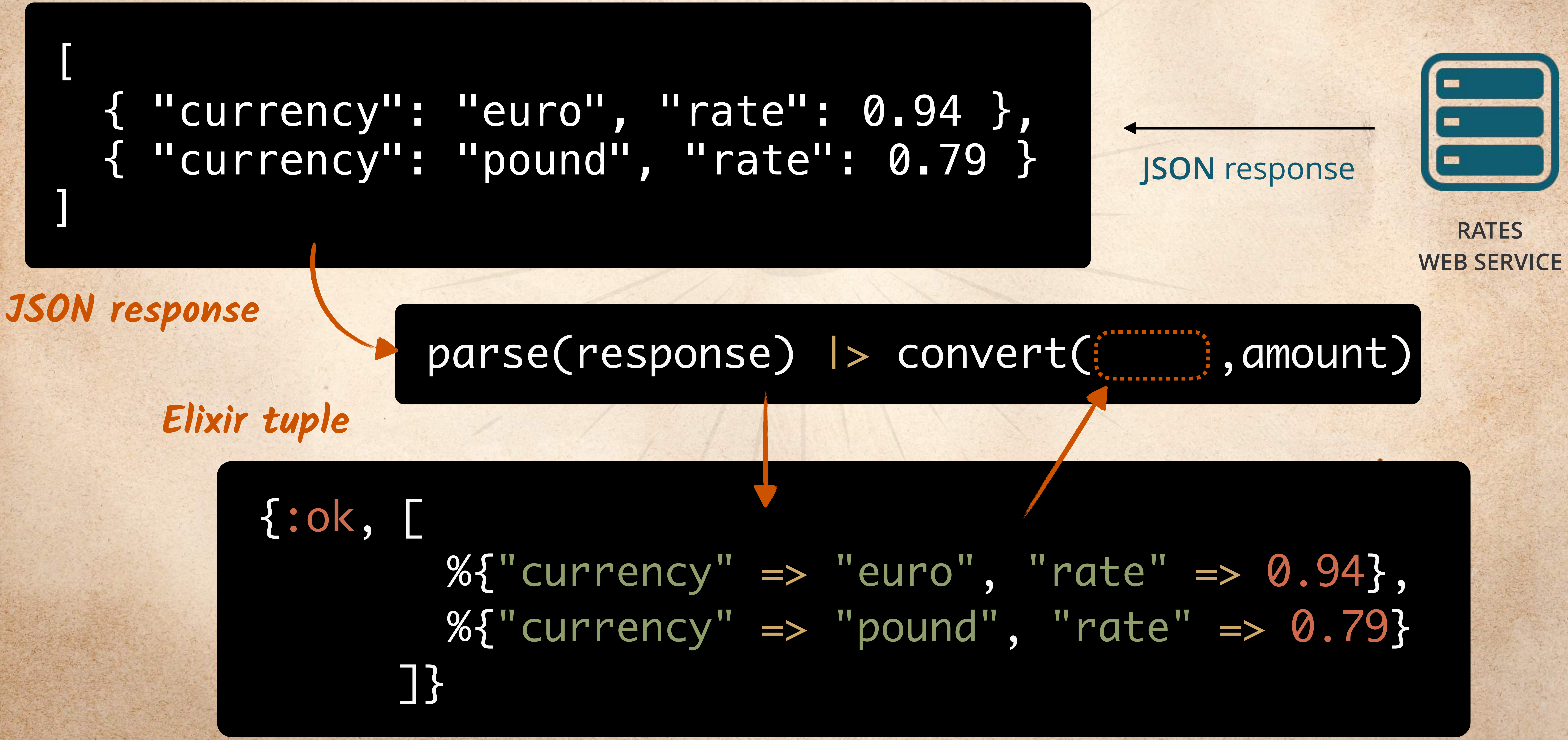
```
defmodule Budget.Conversion do
  ...
  defp parse(%{status_code: 200, body: json_response}) do
    Poison.Parser.parse(json_response)
  end
  ...
end
```

Returns a tuple

*defp **means it's a <u>private</u> function, not to be called from outside its enclosing module.***

From JSON to List of Tuples

The parse function converts the JSON response from the remote server to a tuple, and passes it as the first argument to the convert function.



```
[
  { "currency": "euro", "rate": 0.94 },
  { "currency": "pound", "rate": 0.79 }
]
```

```
parse(response) |> convert(      ,amount)
```

```
{:ok, [
        %{"currency" => "euro", "rate" => 0.94},
        %{"currency" => "pound", "rate" => 0.79}
      ]}
```

Finding Rates and Converting

The convert function grabs the list of tuples via pattern matching and calls find_euro to find the rate for € euro. Lastly, it performs the conversion operation.

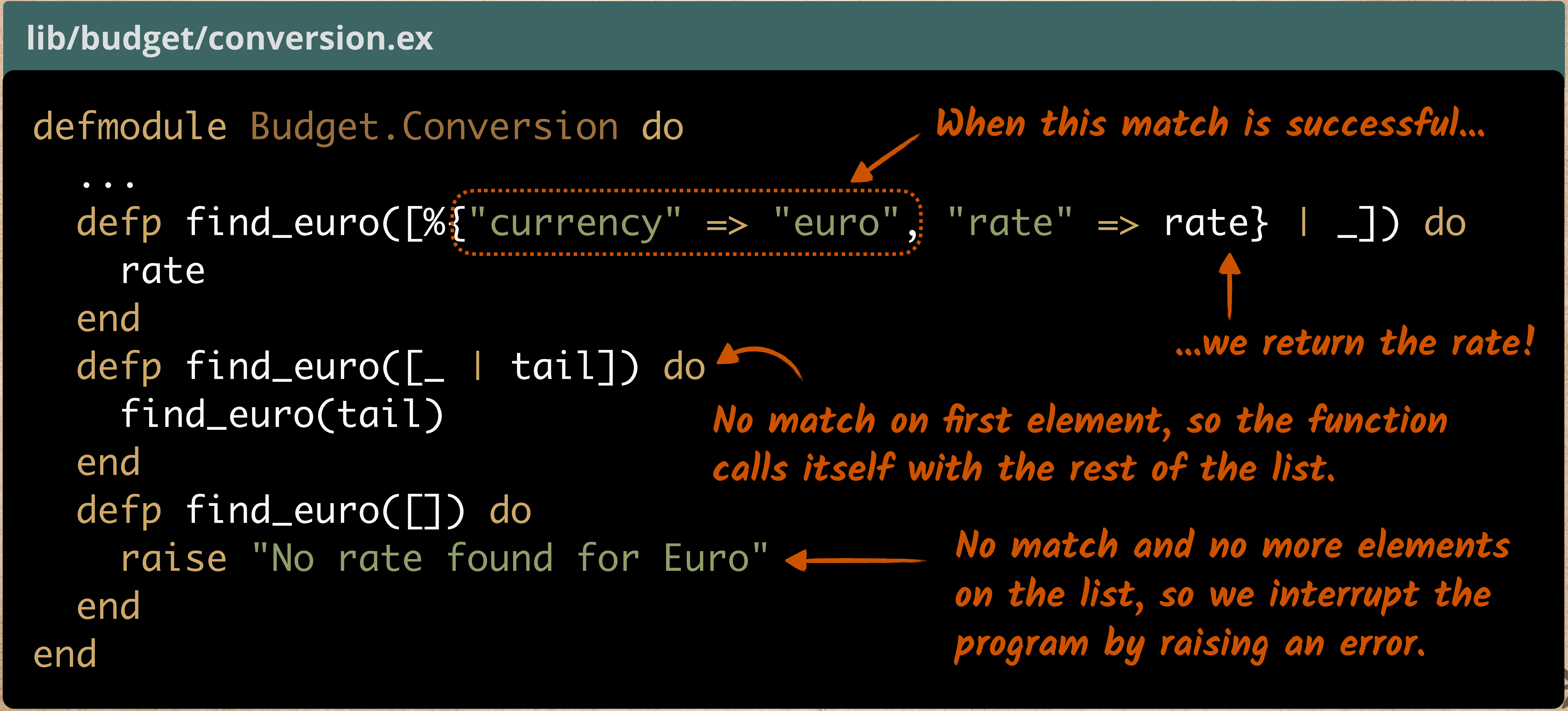
lib/budget/conversion.ex

```
defmodule Budget.Conversion do
  ...
  defp convert({:ok, rates}, amount) do
    rate = find_euro(rates)
    amount / rate
  end
  ...
end
```

Pattern matching

Using Recursion to Find the Rate

We'll use pattern matching and recursion to find the rate for € euro from the list of all rates available.



lib/budget/conversion.ex

```
defmodule Budget.Conversion do
  ...
  defp find_euro([%{"currency" => "euro", "rate" => rate} | _]) do
    rate
  end
  defp find_euro([_ | tail]) do
    find_euro(tail)
  end
  defp find_euro([]) do
    raise "No rate found for Euro"
  end
end
```

When this match is successful...

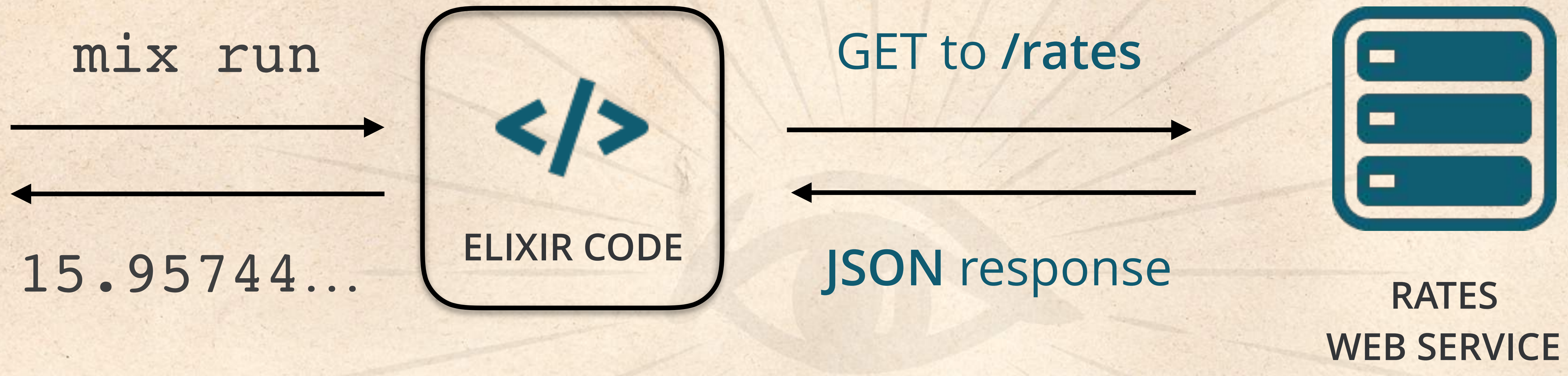
...we return the rate!

No match on first element, so the function calls itself with the rest of the list.

No match and no more elements on the list, so we interrupt the program by raising an error.

Running the Complete Program

We can run the program using mix run and see the expected results printed to the screen.

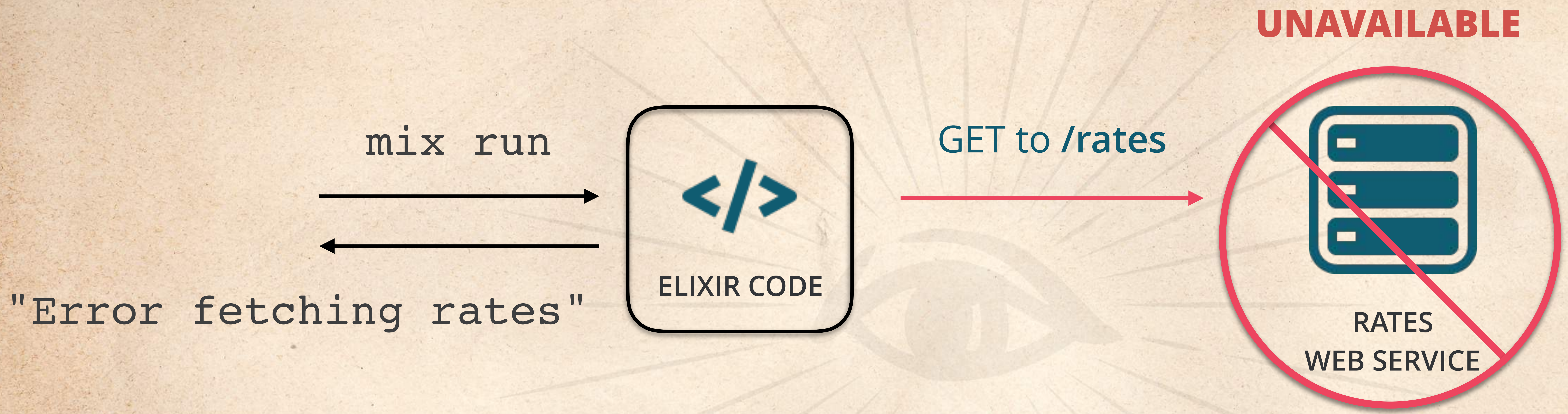


```
$ mix run -e "Budget.Conversion.from_euro_to_dollar(15) |> IO.puts"
```

```
15.957446808510639
```

Running With the Rates Web Service Down

If the rates web service is unavailable, running the program prints the friendly error message.



```
$ mix run -e "Budget.Conversion.from_euro_to_dollar(15) |> IO.puts"
```

```
Error fetching rates
```