Elixir

Programación más funcional que nunca

Semana 1 – MC #1

SofkaU

#ElDesafíoEsContigo

Temas

01 Presentación

02 Erlang y Elixir

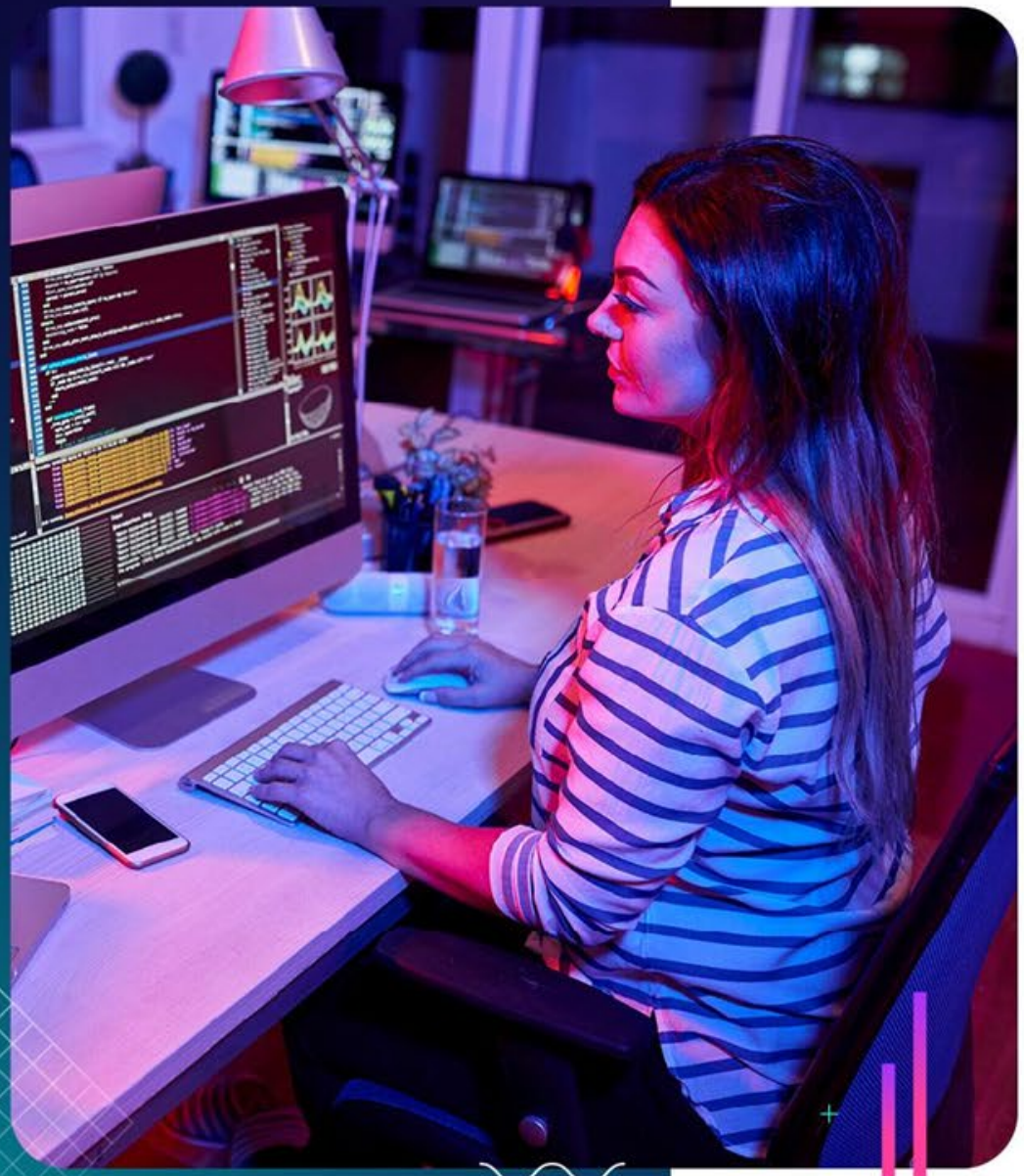
03 Mix

04 Variables, tipos de datos, operadores lógicos/matemáticos

05 Manejo de Strings

Objetivos del curso

- ✓ Comprender los aspectos claves de la programación funcional con Elixir.
- ✓ Comprender los fundamentos y las prácticas relacionadas a la programación con Elixir.
- ✓ Aplicar de manera correcta las mejores prácticas a la hora de programar APIs



¿A quién va dirigido este curso?

- Personas con conocimiento en por lo menos 1 lenguaje de programación
- Mente abierta a los cambios
- Cierta experiencia programando

Detalles

Aprobación:
- *Todos los assessments mínimo 75%*
- *Asistencia mínima 62,5% (5 de 8 sesiones)*

Modalidad: Semi-asistido

Duración: 5 semanas

Inicio del curso: 29 de mayo
Fin del curso: 3 de julio

4 tipos de sesiones: Masterclasses, Workshops Coaching y Assessments.

Masterclass: Lunes y Martes
Hora inicio: 07H15
Hora fin: 08H15
**Serán grabadas*

Workshop: Lunes y Martes
Hora inicio: 08H45
Hora fin: 10H00

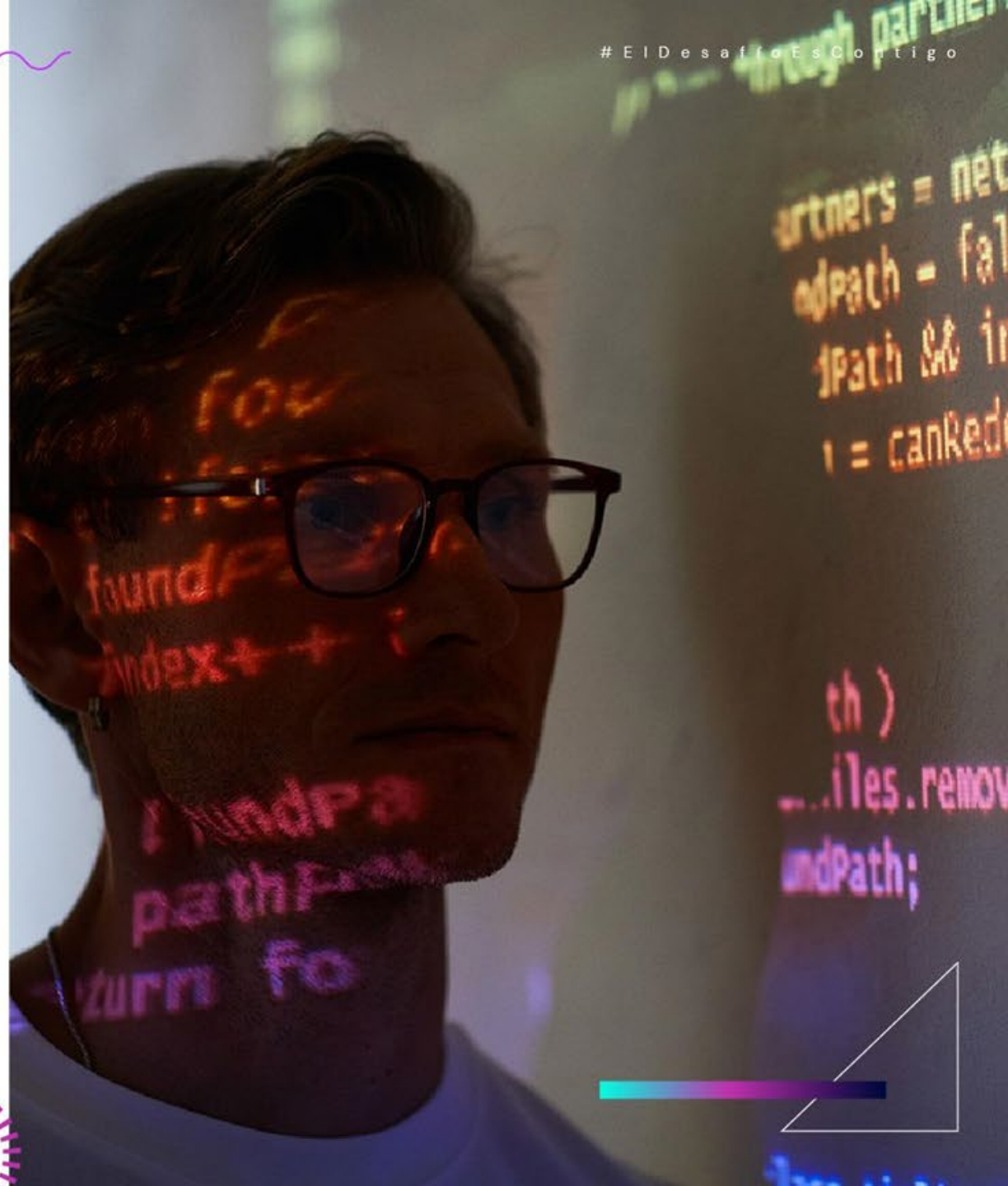
Coaching: Miércoles y Jueves
Hora inicio: 08H00
Hora fin: 14H00

Assessments: Viernes
Hora inicio: 07H30
Hora fin: 10H00

MC #1

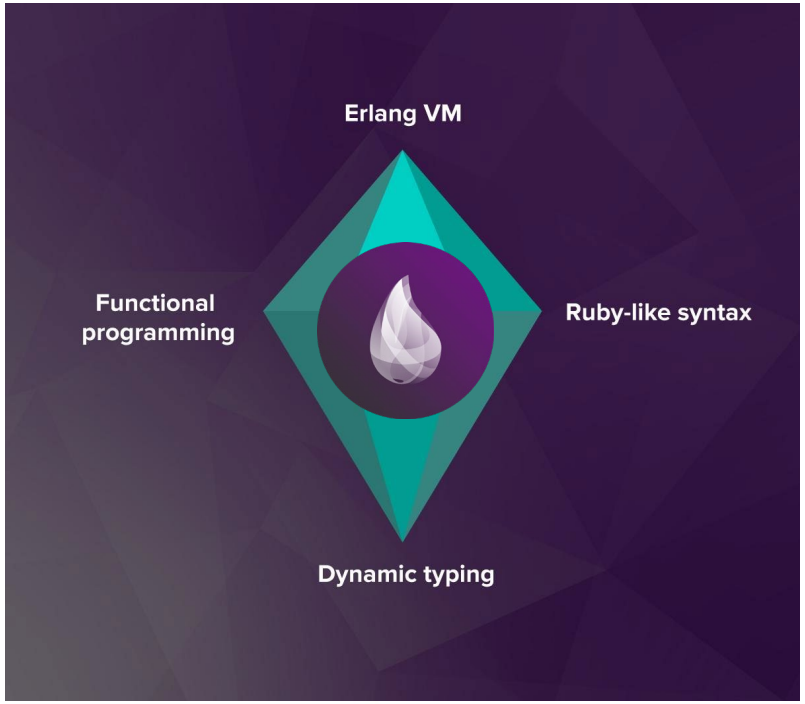
Erlang

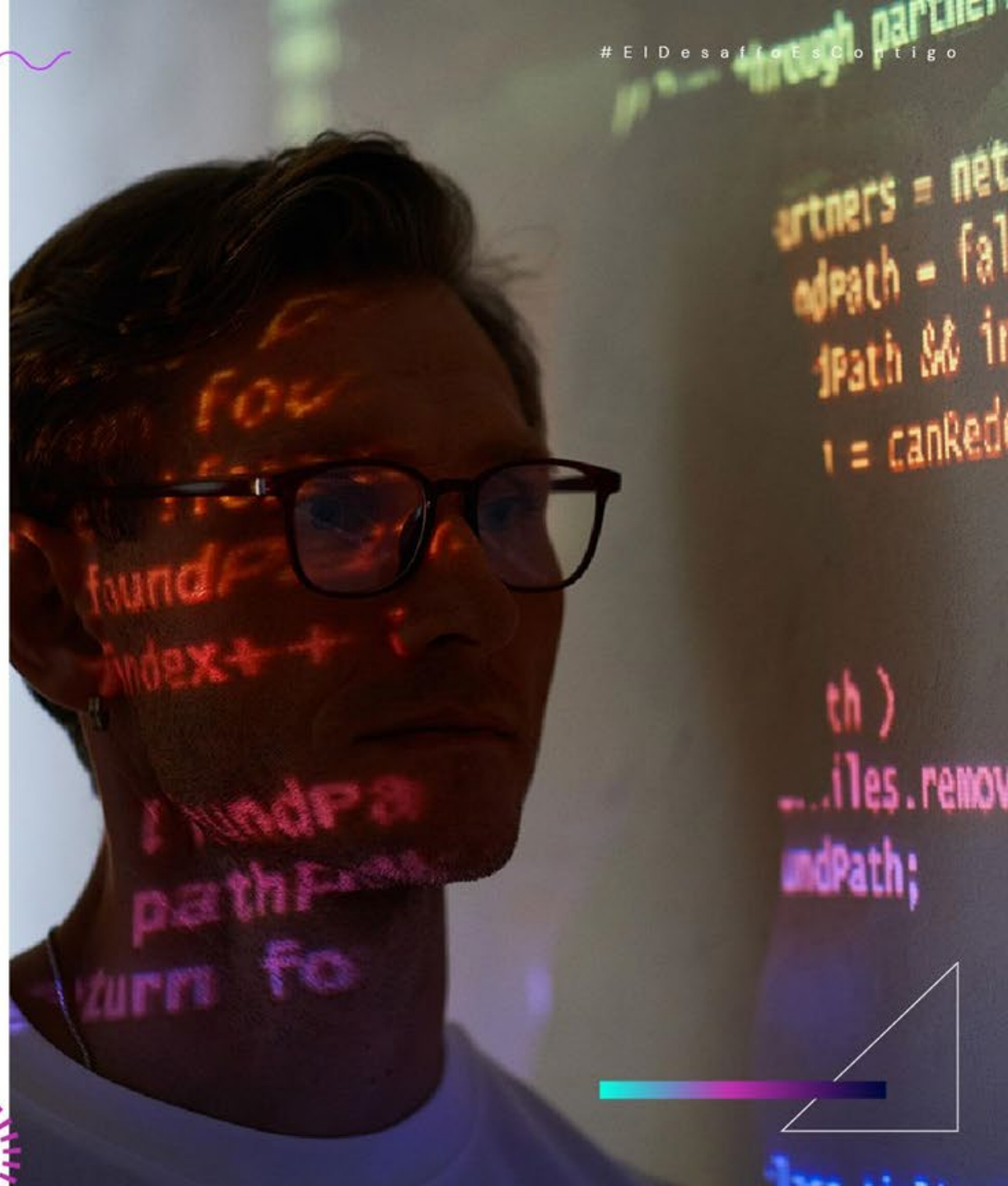
Lenguaje de alto nivel basado en procesos, patrones y paradigma funcional para escribir aplicaciones concurrentes y distribuidas de funcionamiento ininterrumpido mediante procesos concurrentes que forman parte de la estructura



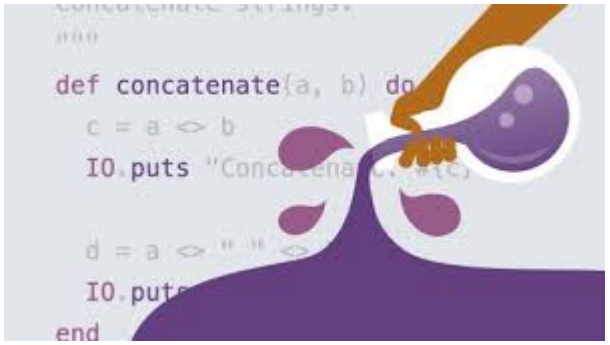
Elixir

Aportando lo mejor de Erlang con una curva de aprendizaje mucho más sencilla e intuitiva, su valor primordial es el manejo de datos de información a gran escala.

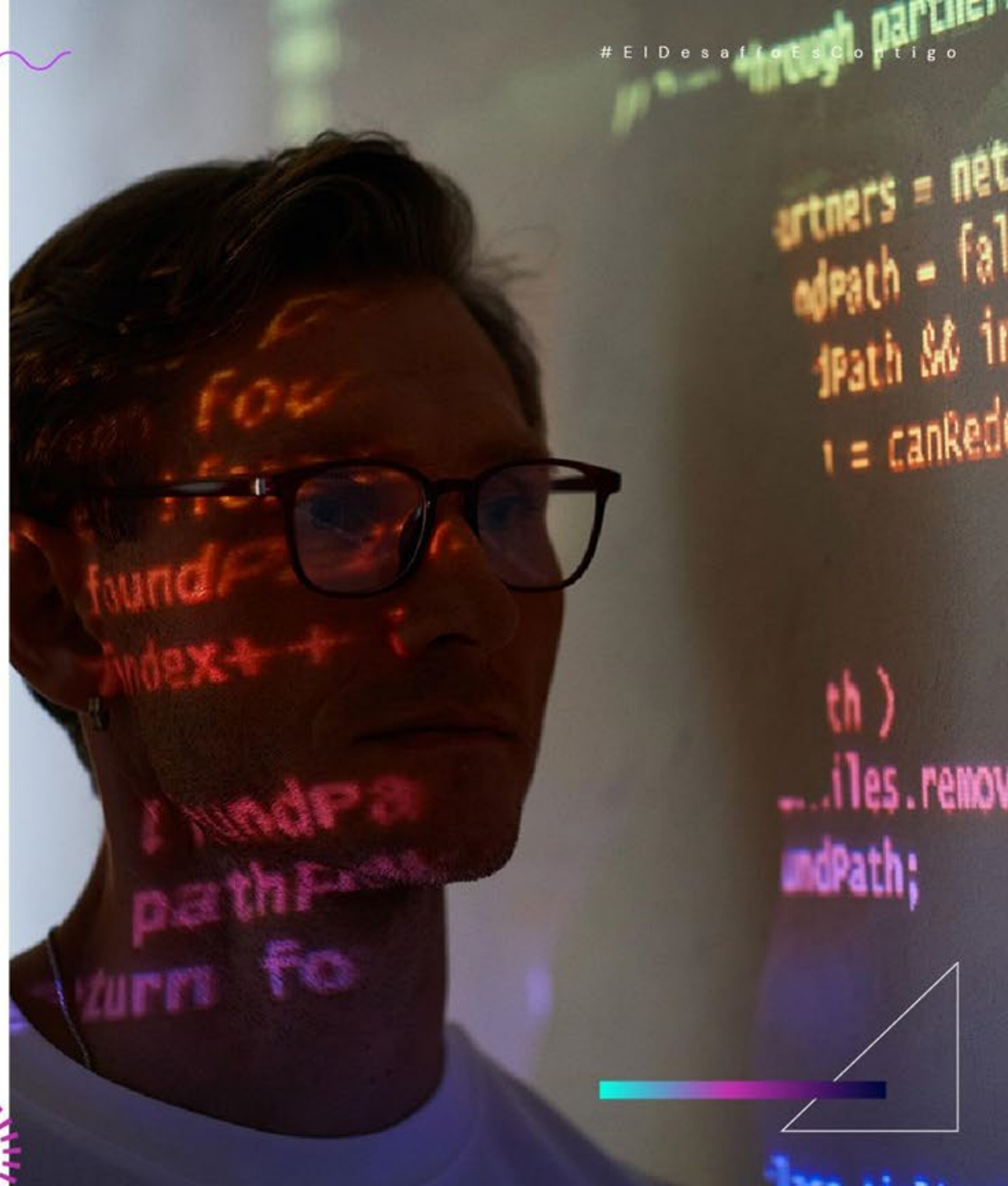




Mix



- Herramienta de compilación y gestor de paquetes para el lenguaje de programación Elixir.
- Mix está integrado en el lenguaje Elixir y viene preinstalado con la distribución de Elixir.
- Proporciona comandos para compilar y ejecutar código Elixir, probar código, gestionar dependencias y generar documentación.

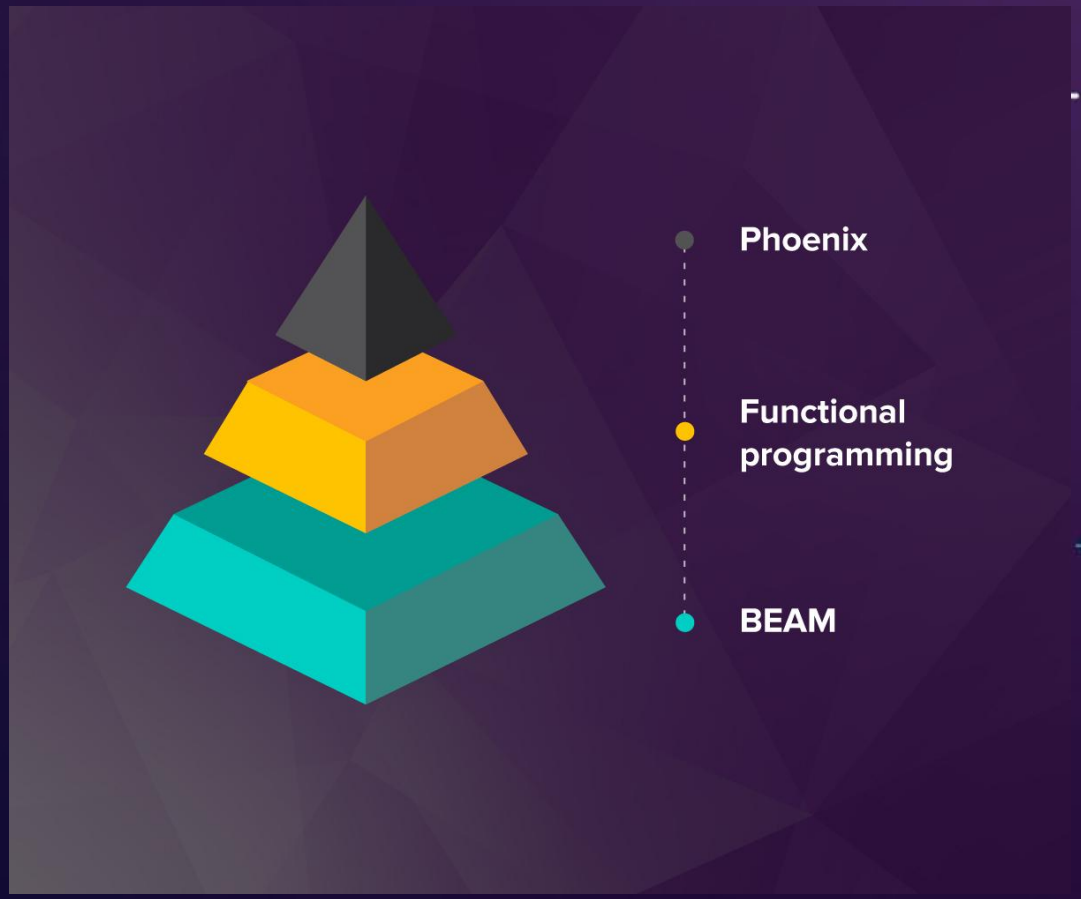


```
Usage: mix [task]

Examples:

    mix             - Invokes the default task (mix run) in a project
    mix new PATH    - Creates a new Elixir project at the given path
    mix help        - Lists all available tasks
    mix help TASK   - Prints documentation for a given task

The --help and --version options can be given instead of a task for usage and versioning information.
```

BEAM

Un componente clave tanto de Erlang como de Elixir, ya que proporciona el entorno de ejecución para estos lenguajes.

Esta máquina virtual es conocida por su alto rendimiento, capacidad de concurrencia y tolerancia a fallos, lo que la hace idónea para crear sistemas distribuidos a gran escala.



Características que hereda de Erlang

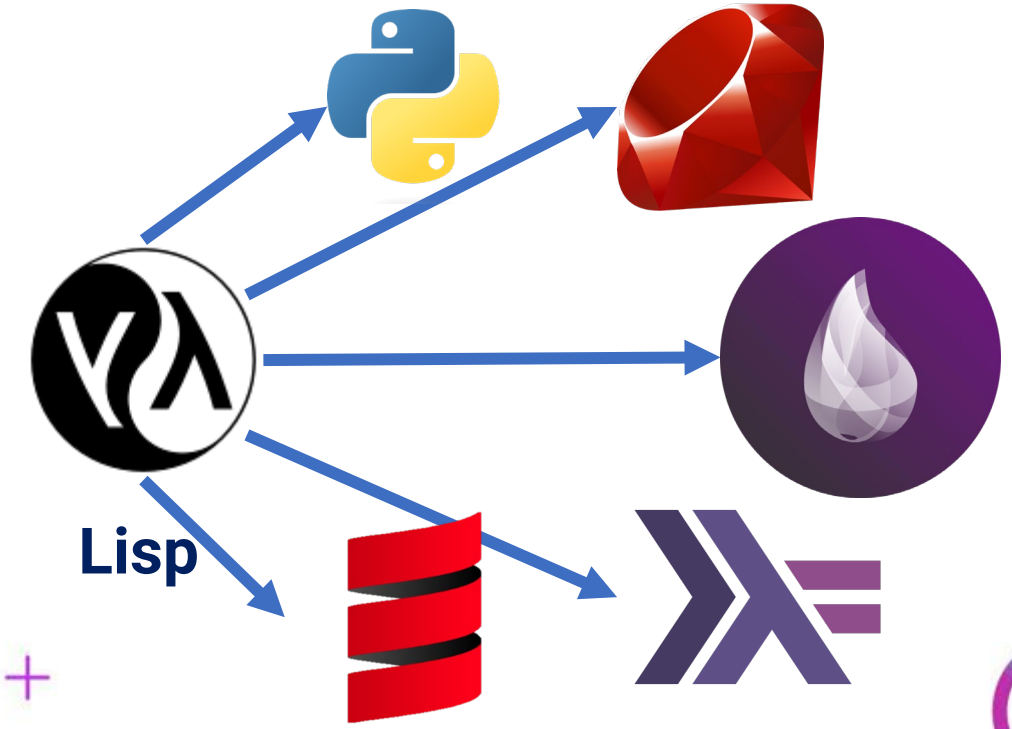
- Programación funcional
- Concurrencia
- Computación distribuida
- OTP

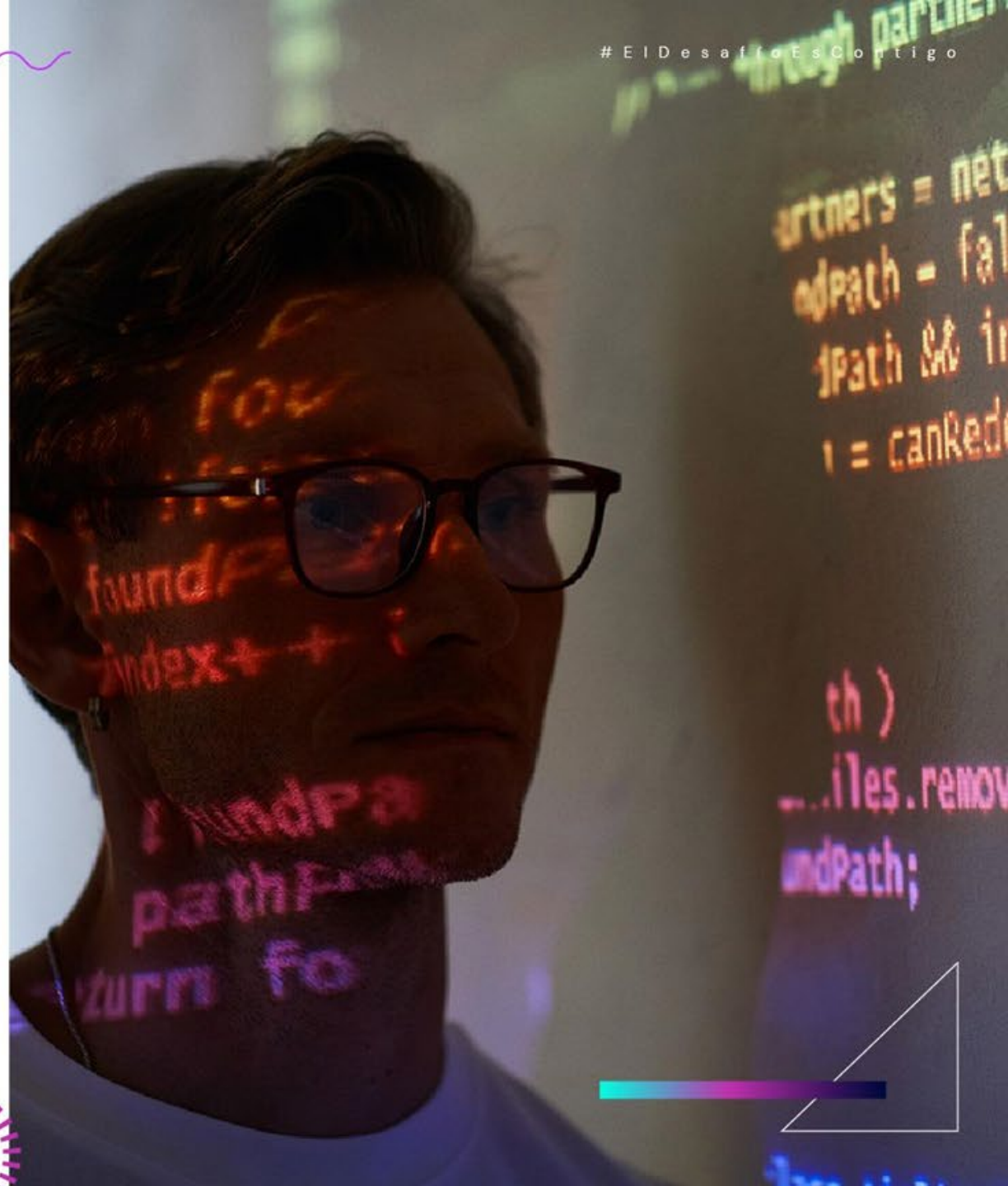
Características para Backend

- Alto rendimiento
- Escalabilidad
- Tolerancia a fallos
- Phoenix
- Ecto
- Plug

REPL

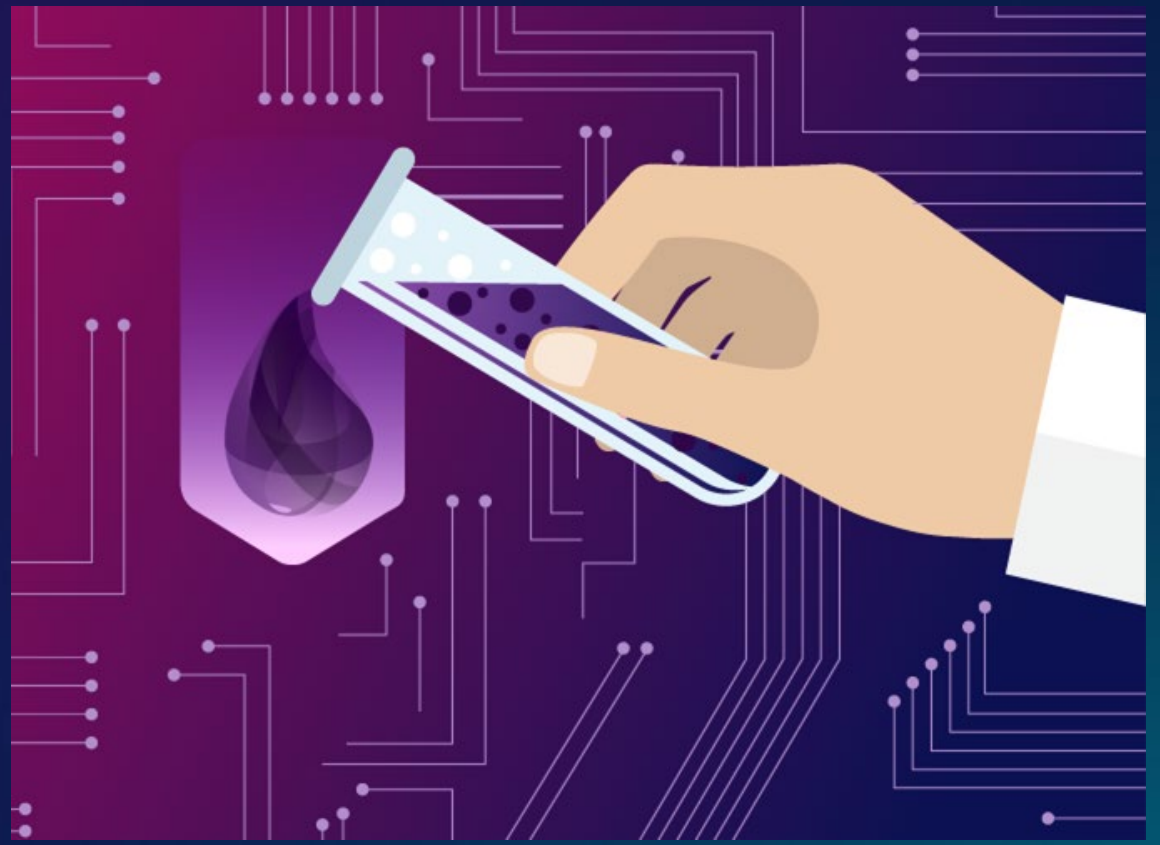
Read-Eval-Print-Loop: Software donde se ejecuta interactivamente código o 'por sobre la marcha' basado en un proceso cíclico donde se lee un fragmento de código ingresado por el usuario, se evalúa y se devuelve un resultado al usuario.

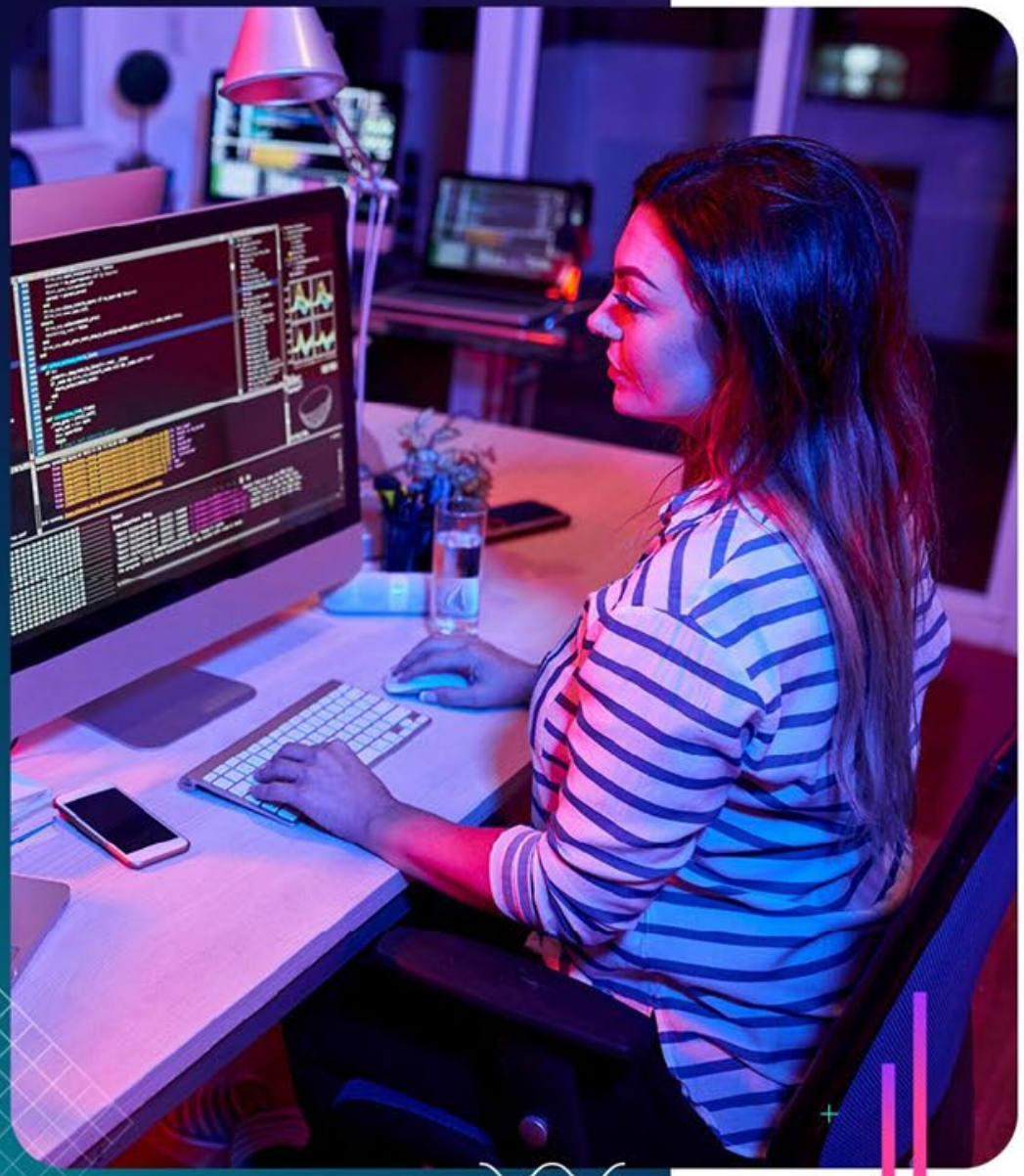




Elixir

Instalación





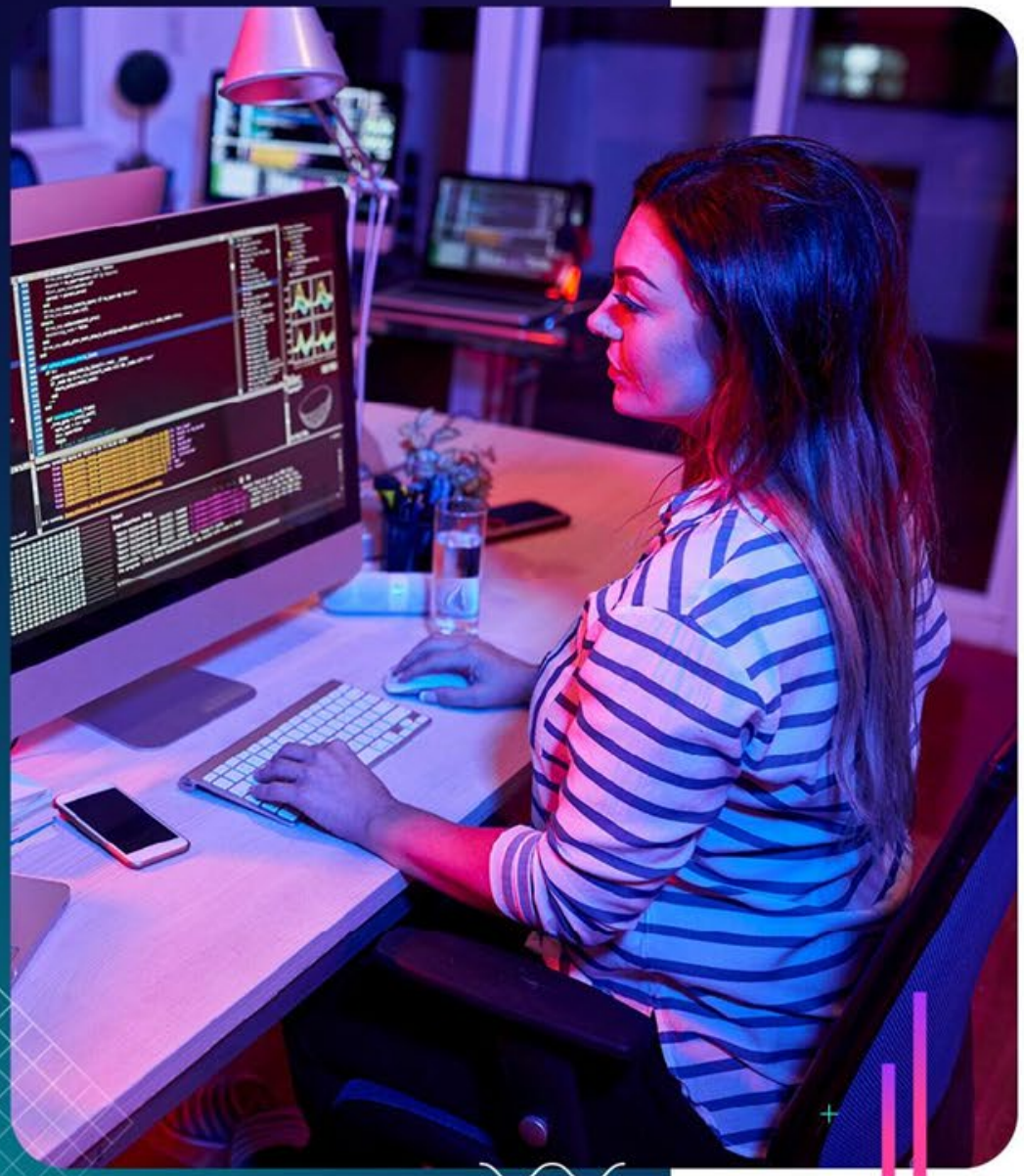
Instalando Elixir

elixir-lang.org/install.html

HOME INSTALL

elixir

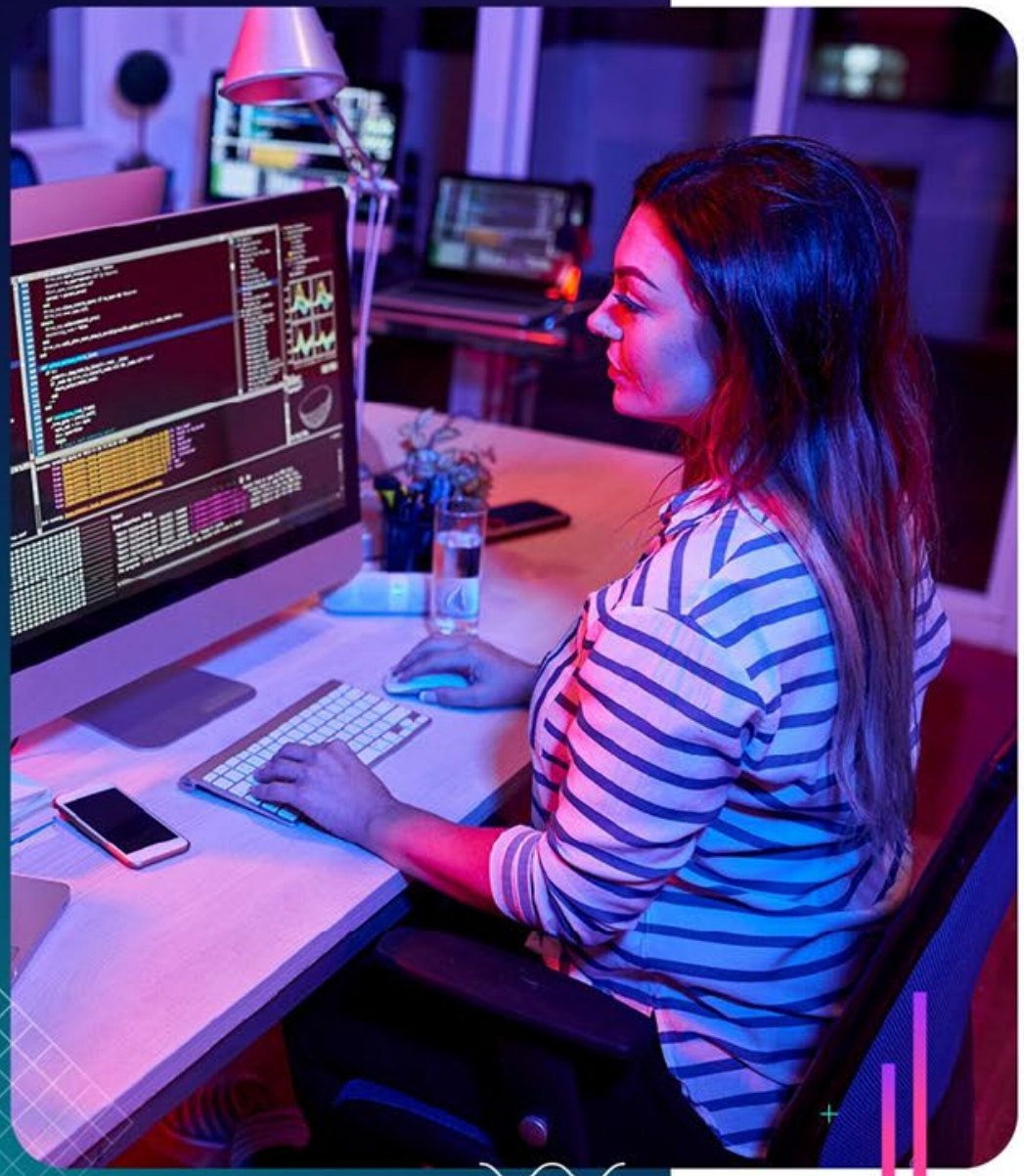
Install



Windows

Windows

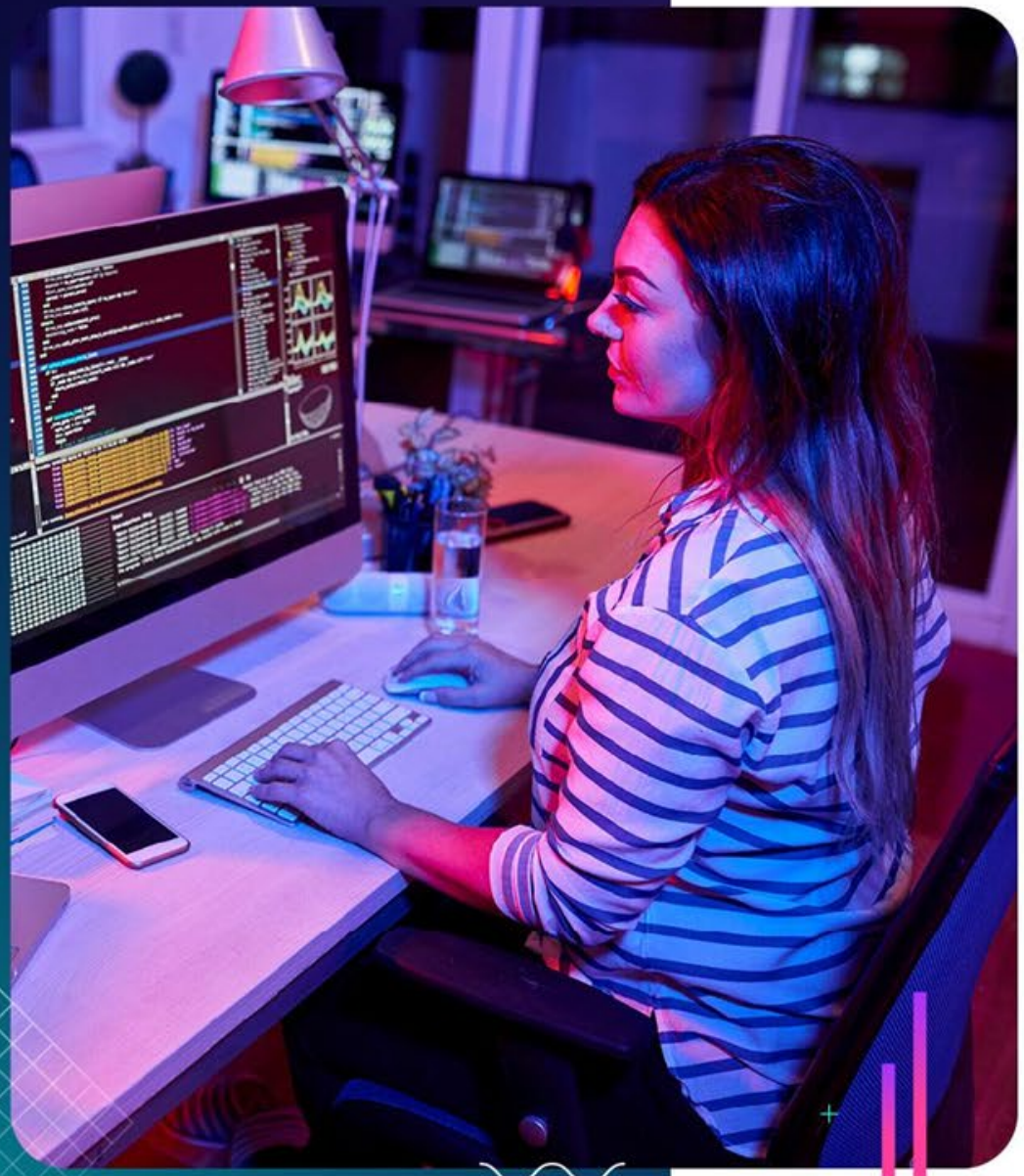
- Using our web installer:
  - Download the installer
  - Click next, next, ..., finish
  - If you run into issues, check out the Windows Installer issues tracker
- Using Scoop:
  - Install Erlang: `scoop install erlang`
  - Install Elixir: `scoop install elixir`



macOS

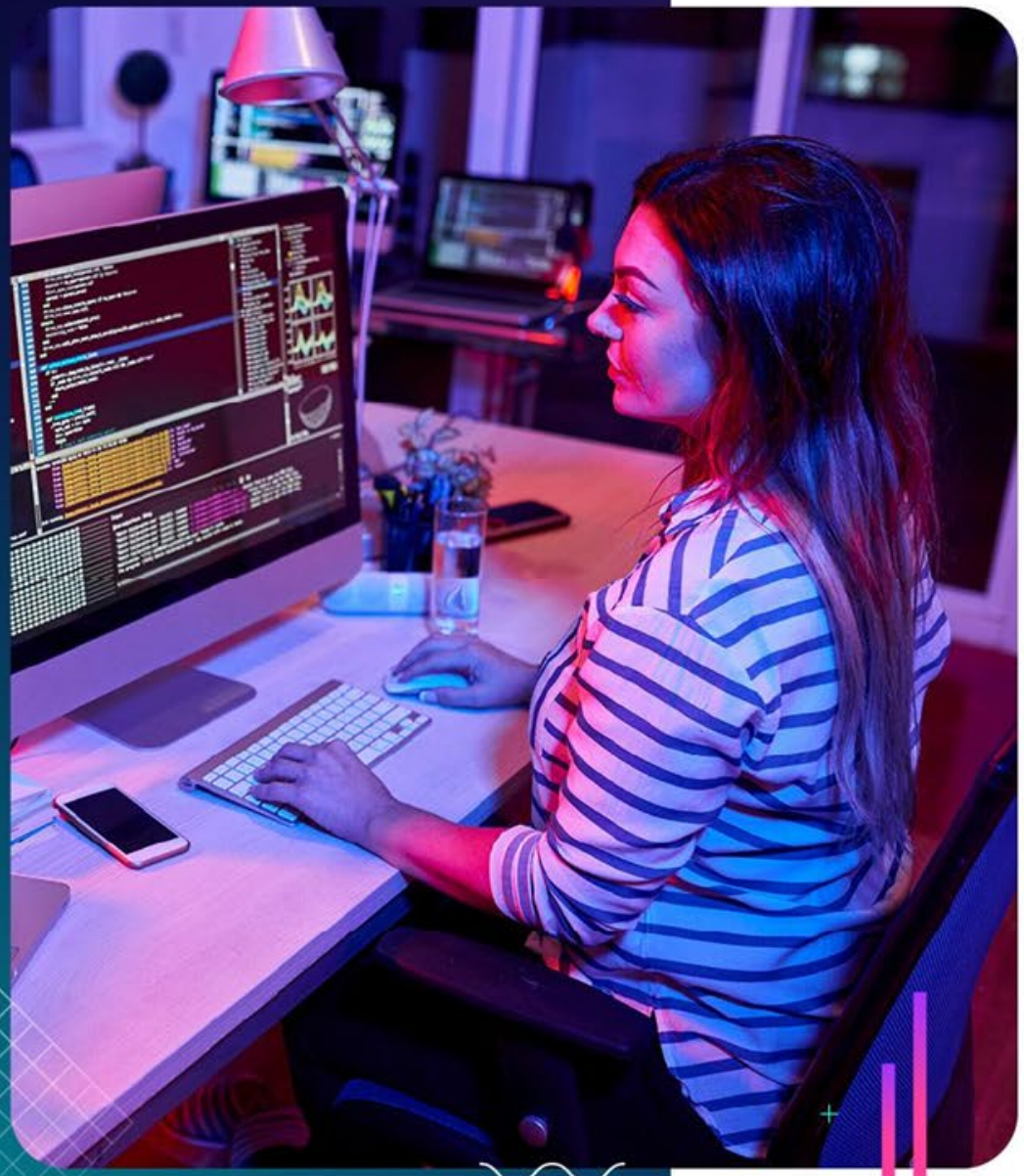
macOS

- Using Homebrew:
  - Run: `brew install elixir`
- Using Macports:
  - Run: `sudo port install elixir`

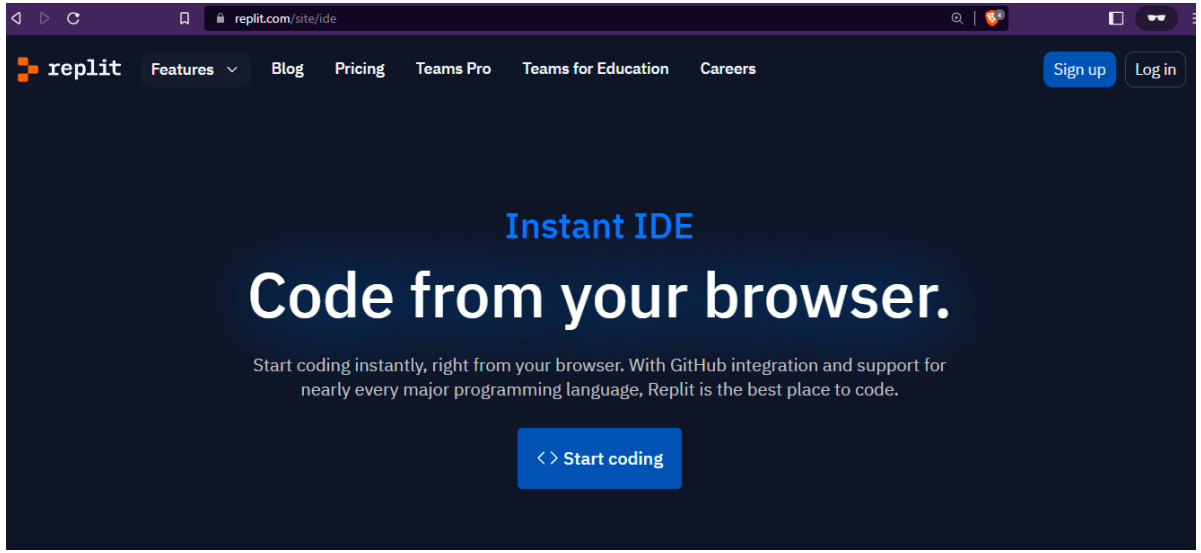


Linux

- **Alpine Linux** (Community repository)
  - Run: `apk add elixir`
- **Arch Linux** (Community repository)
  - Run: `pacman -S elixir`
- **Debian**
  - *See below the instructions for Ubuntu*
- **Fedora 21 (and older)**
  - Run: `yum install elixir`
- **Fedora 22 (and newer)**
  - Run `dnf install elixir erlang`
- **Gentoo**
  - Run: `emerge --ask dev-lang/elixir`
- **GNU Guix**
  - Run: `guix package -i elixir`
- **openSUSE (and SLES)**
  - Add Elixir/Erlang repository: `zypper ar -f obs://devel:languages:erlang/Elixir-Factory`
  - Run: `zypper in elixir`
  - Optional: if you want to use the latest Erlang, you can use this repository: `zypper ar -f obs://devel:languages:erlang:Factory Erlang-Factory`
- **Slackware**
  - Using Sbopkg:
    - Run: `sbopkg -ki "erlang-otp elixir"`
  - Manually:
    - Download, build and install from SlackBuilds.org: `erlang-otp`, and `elixir`
- **Solus**
  - Run: `eopkg install elixir`
- **Ubuntu** or **Debian**
  - From primary package repositories:
    - Run: `sudo apt-get install elixir`
  - From Erlang Solutions, for more recent Elixir/Erlang versions on Ubuntu LTS (< 22.04) or Debian Stable releases:
    - Add Erlang Solutions repository: `wget https://packages.erlang-solutions.com/erlang-solutions_2.0_all.deb && sudo dpkg -i erlang-solutions_2.0_all.deb`
    - Run: `sudo apt-get update`
    - Install the Erlang/OTP platform and all of its applications: `sudo apt-get install esl-erlang`
    - Install Elixir: `sudo apt-get install elixir`
- **Void Linux**
  - Run: `xbps-install -S elixir`

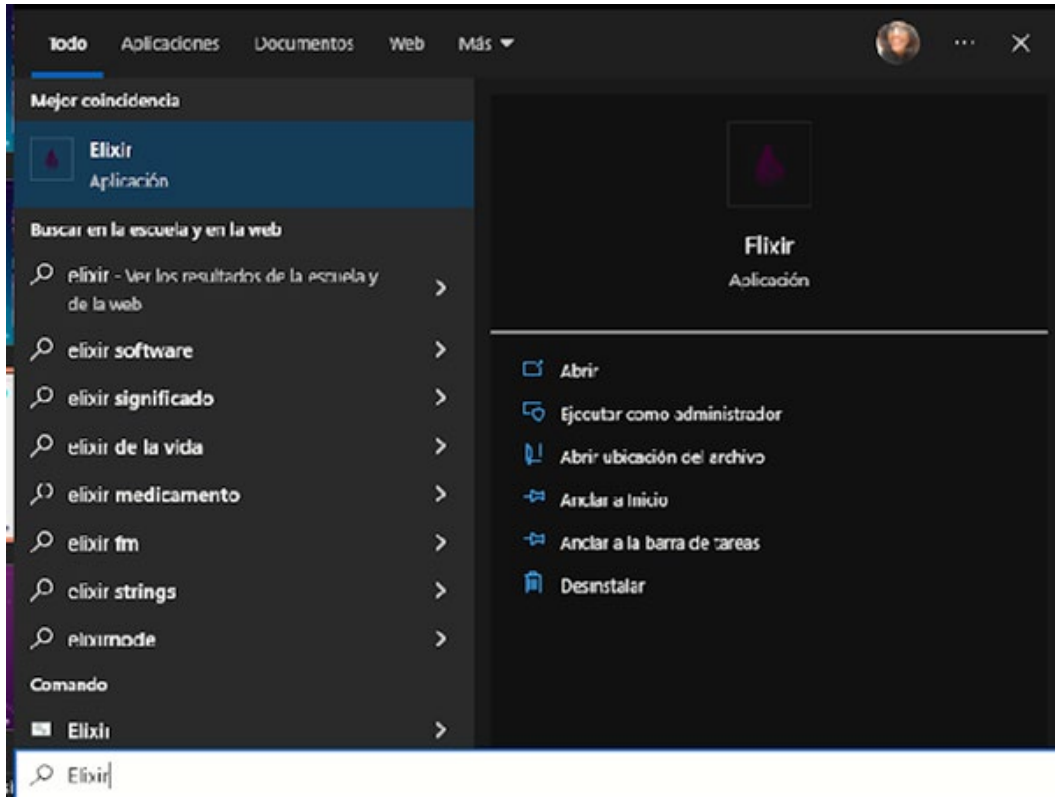


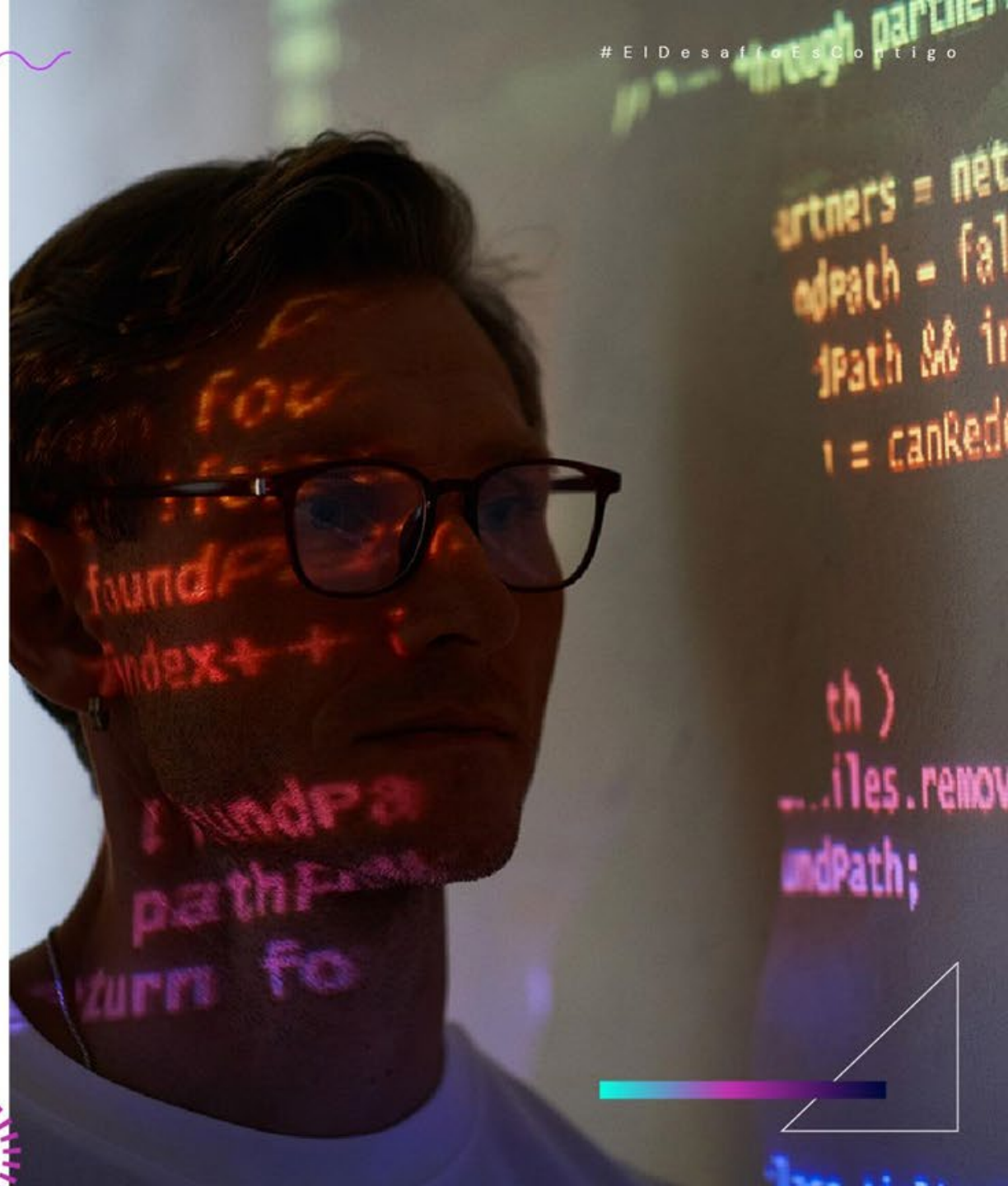
Opción online:



Llamando al REPL de Elixir: iex

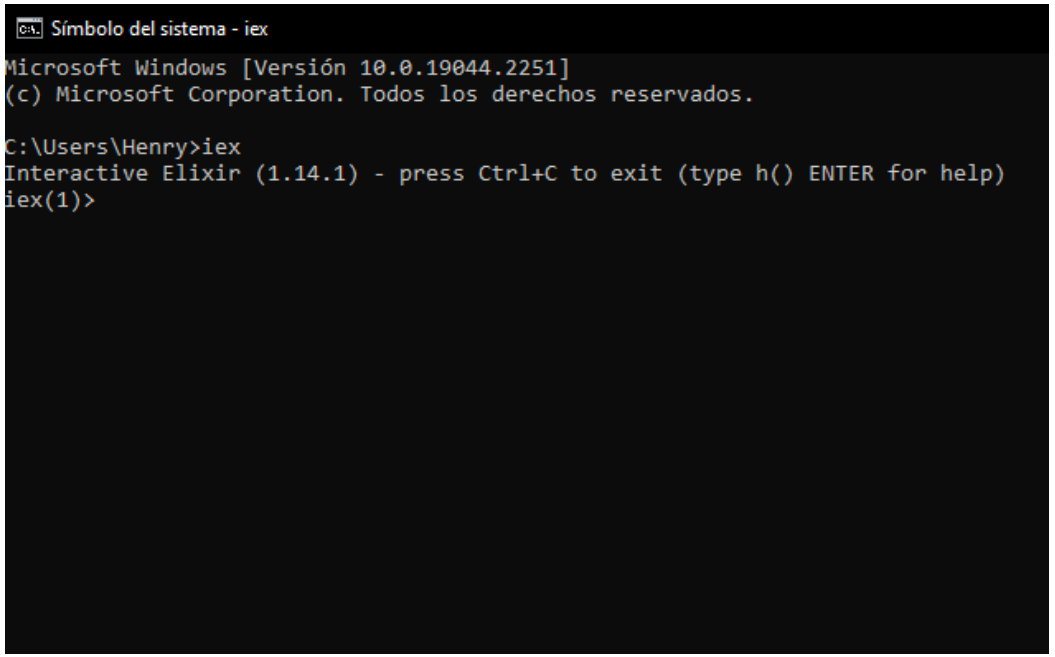
Opción 1:



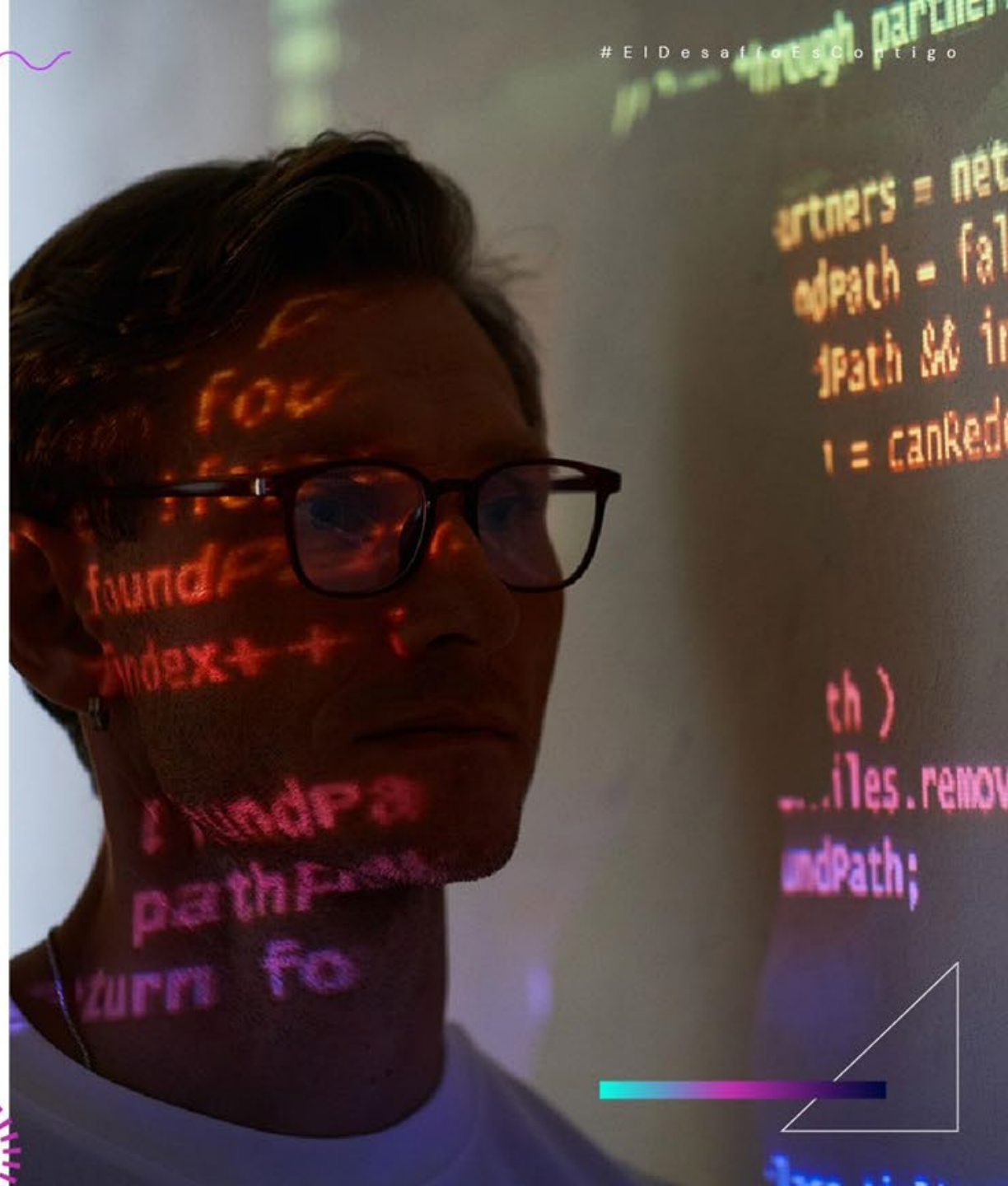


Llamando al REPL de Elixir: iex

Opción 2:

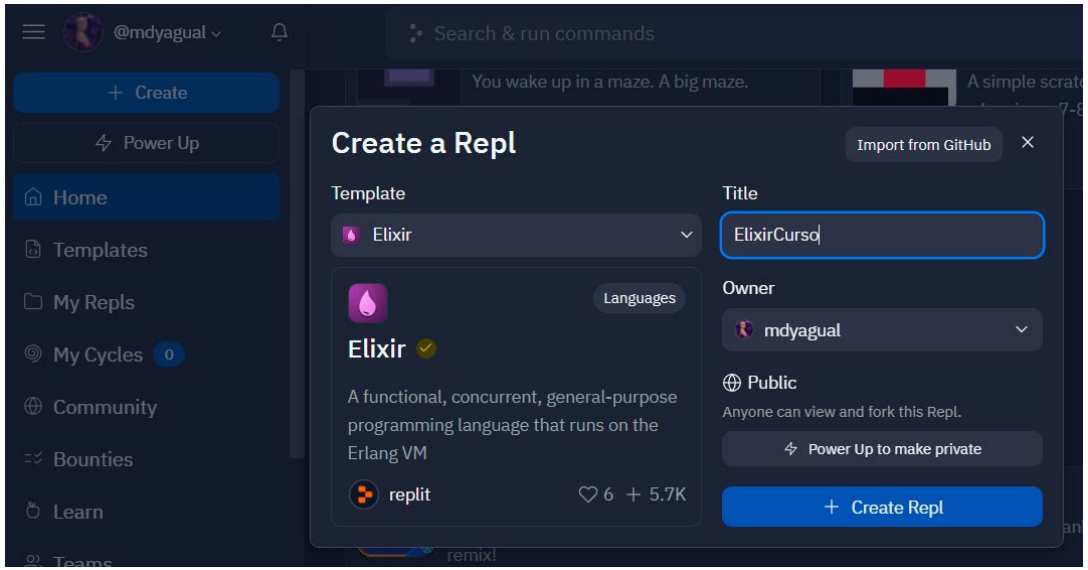


"Comprobemos su funcionalidad usándolo como una calculadora básica."

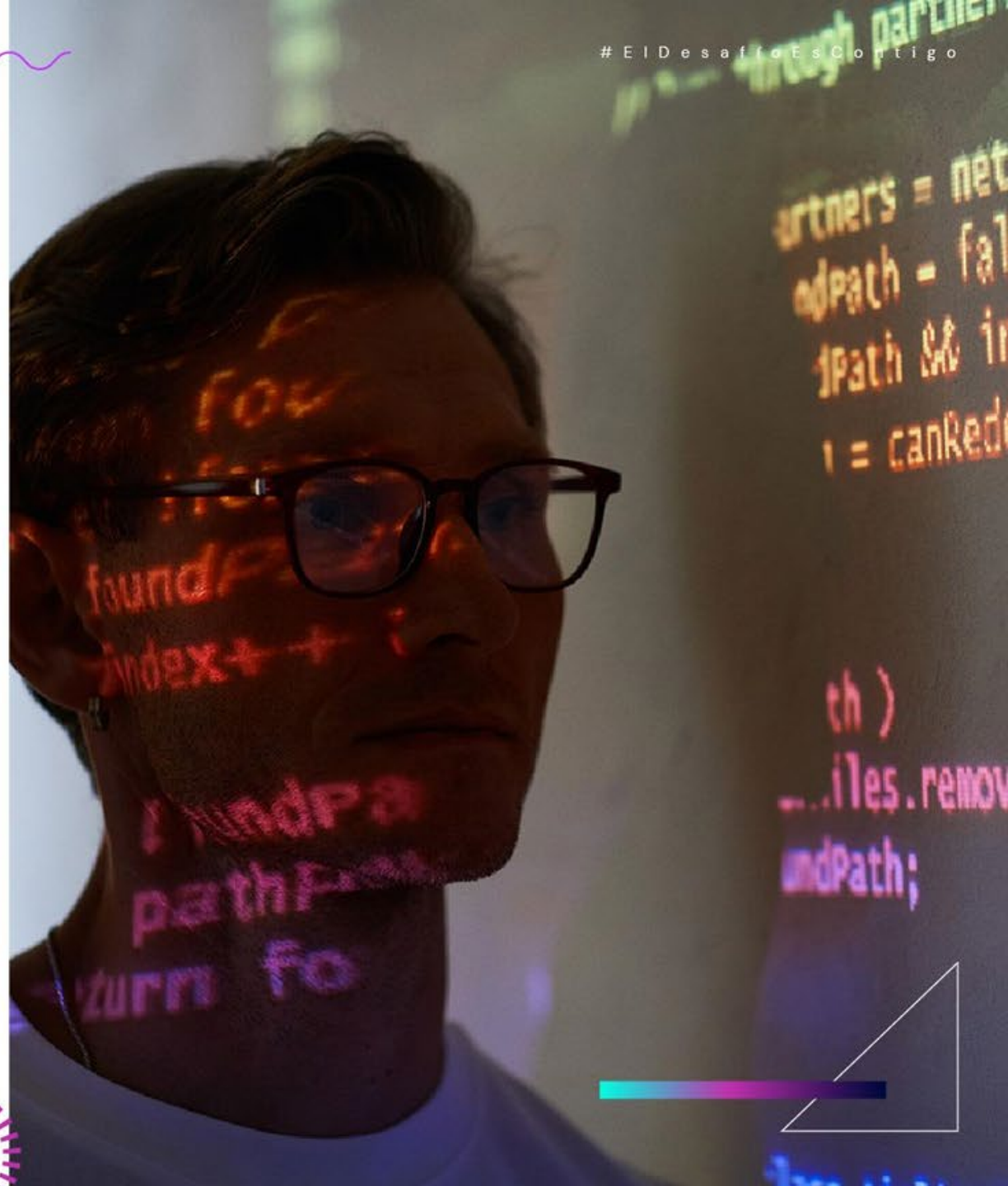


Usando Repl.it

Opción 3:

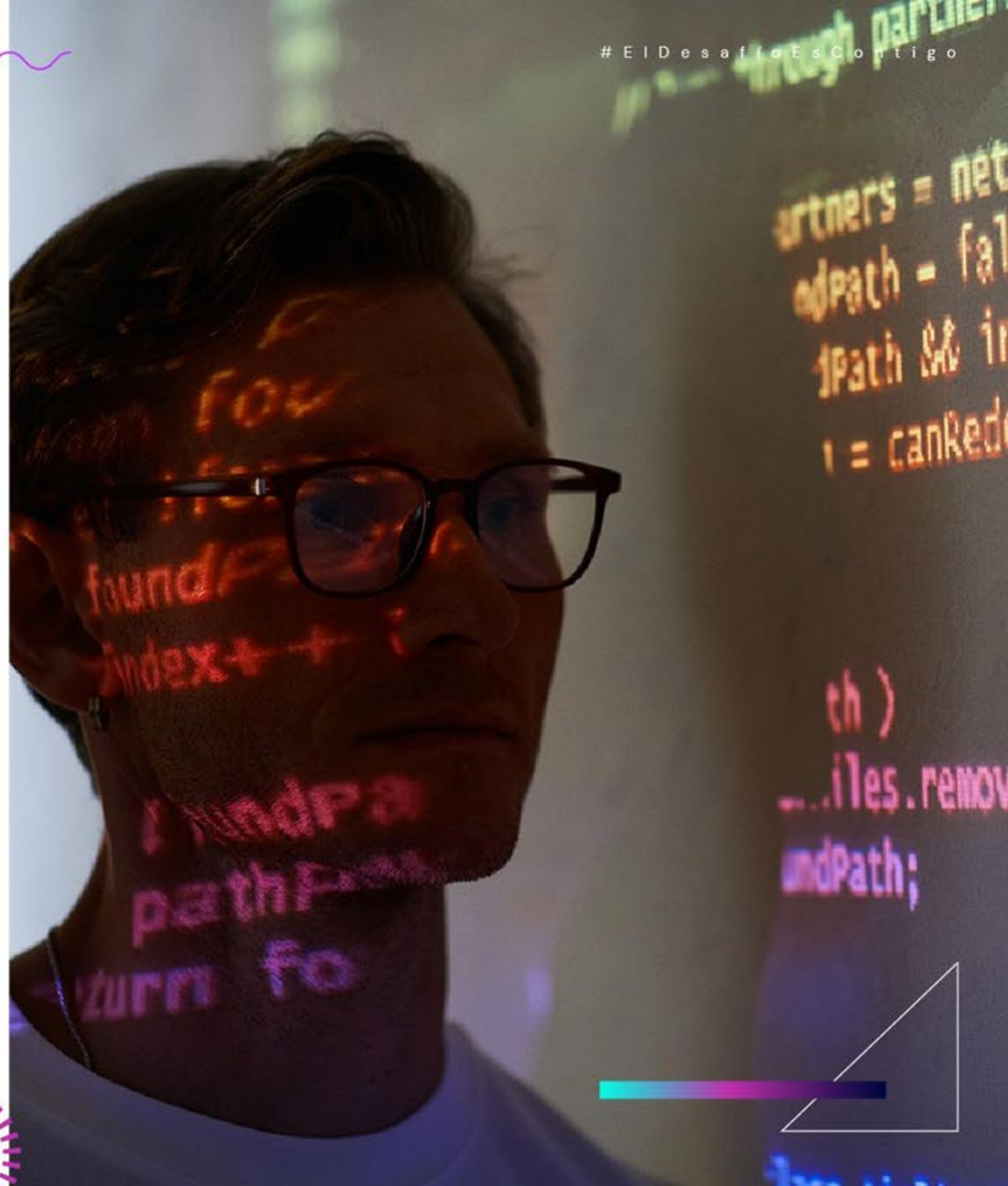


"Comprobemos su funcionalidad usándolo como una calculadora básica."



Requerimientos

- **IDE de desarrollo:** IntelliJ, VSC u otro que reconozca la sintaxis de Elixir.
- Elixir **v1.14.4**
- PostgreSQL **v14.7** en adelante - Local o dockerizado
- Phoenix **v1.7.2**



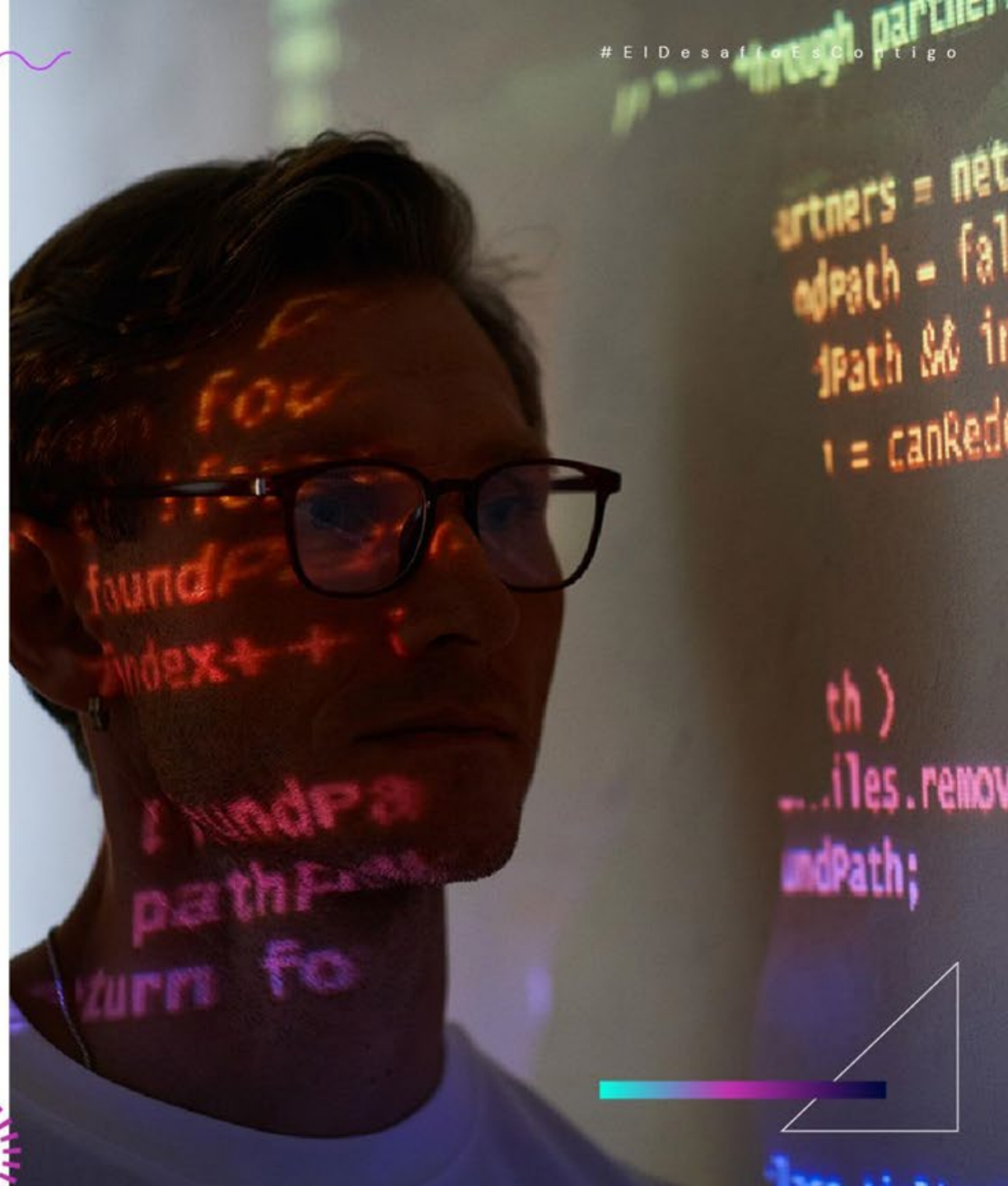
Tipos de datos

Primitivos

- Integer: representa un número entero (por ejemplo, 1, 2, 3).
- Float: representa un número decimal (por ejemplo, 3,14, 0,5).
- Atom: representa una constante con un nombre (por ejemplo, :ok, :error).
- Boolean: representa verdadero o falso.
- Nil: representa la ausencia de un valor.

Estructurados

- Tupla: una colección de elementos con un tamaño y un orden fijos (por ejemplo, {1, 2, 3}).
- Lista: colección de elementos de tamaño y orden variable (por ejemplo, [1, 2, 3]).
- Mapa: colección de pares clave-valor (por ejemplo, %{nombre: "Alicia", edad: 30}).
- Struct: un mapa con un conjunto predefinido de claves y valores predeterminados (por ejemplo, defstruct nombre: "Alicia", edad: 30).

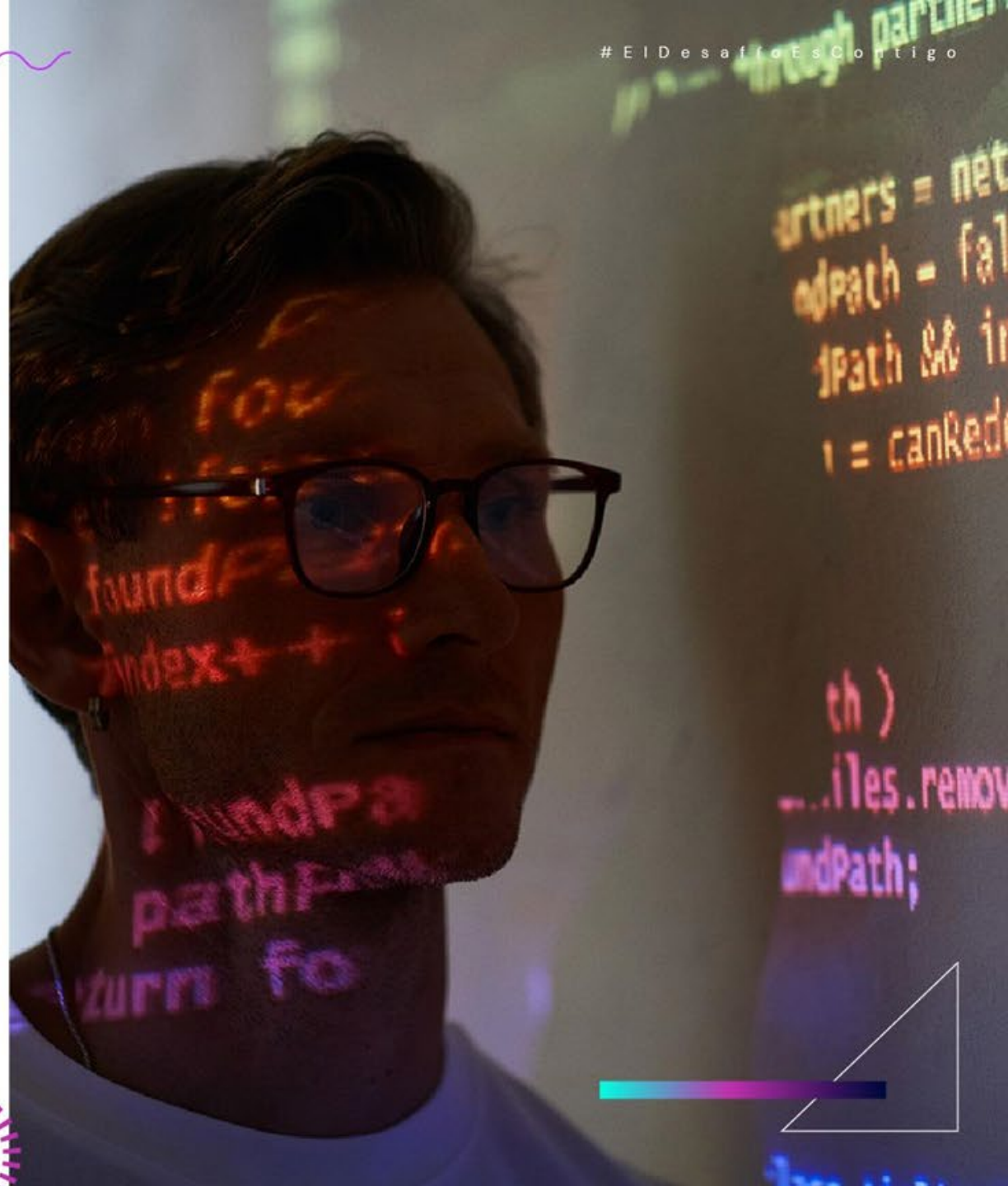


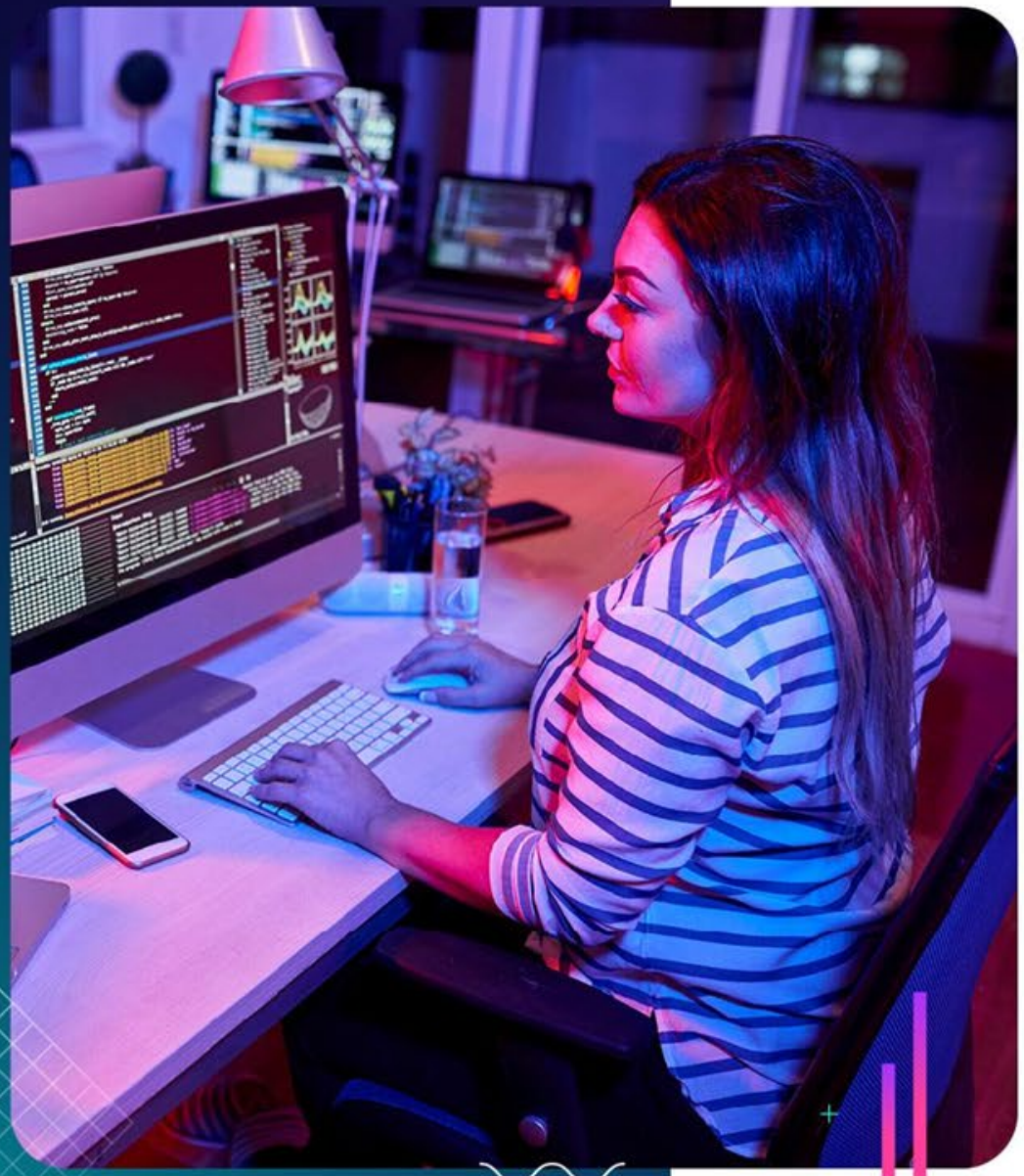
Mathematical Operators

+	Adds left and right
-	Subtracts the right from left
*	Multiplies left and right
/	Divides left by right

Comparison Operators

===	Left and right are same value **and** data type?
!==	Left and right are **not** same value **and** data type?





Expresiones booleanas

Estrictas

- Los operadores booleanos and, or y not siempre evalúan ambos lados de la expresión, independientemente del valor del primer operando.

```
false and raise "Error"
```

No estrictas

- Los operadores && y || tienen un comportamiento de cortocircuito, lo que significa que sólo evalúan el segundo operando si es necesario.

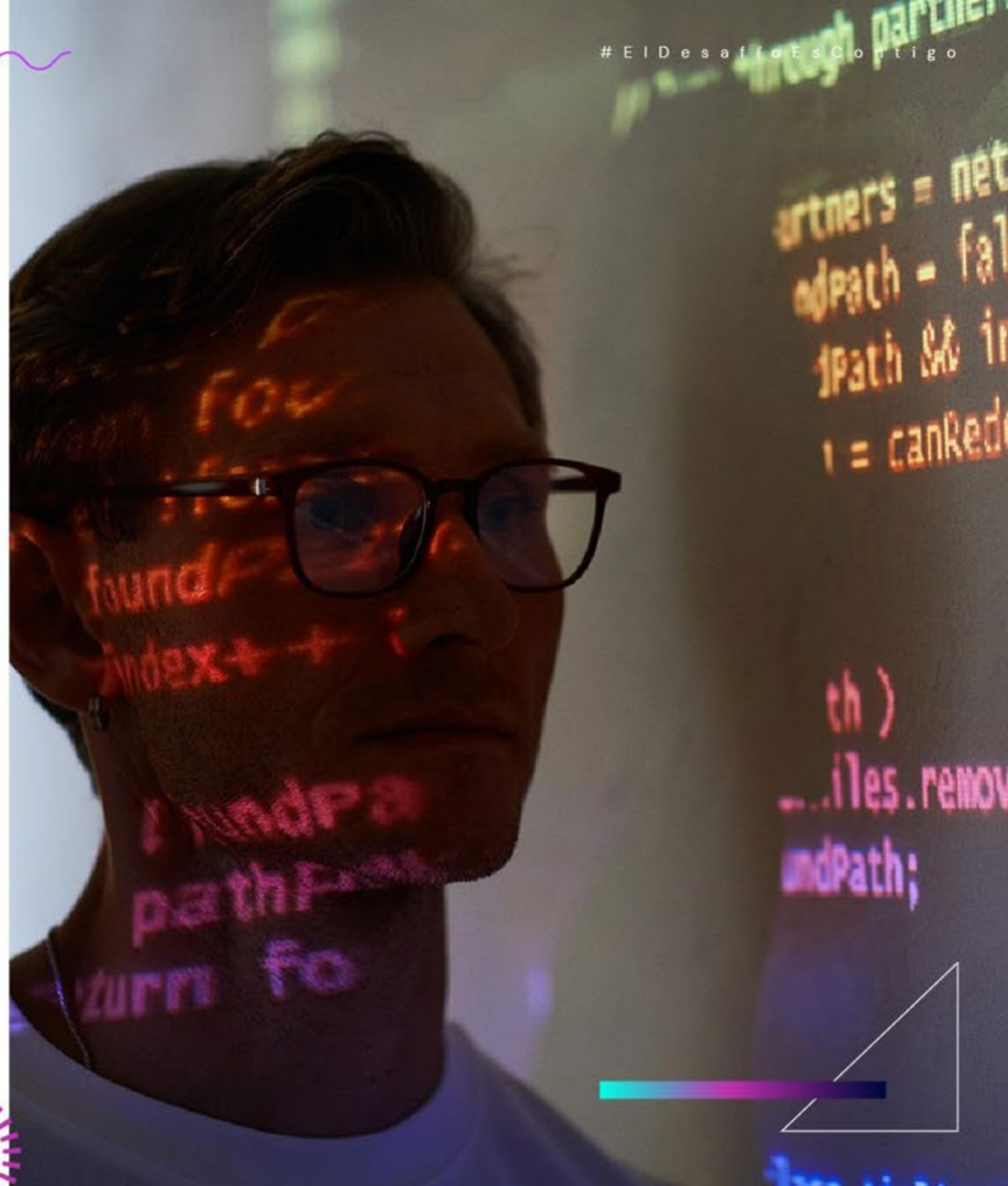
```
false && raise "Error"
```

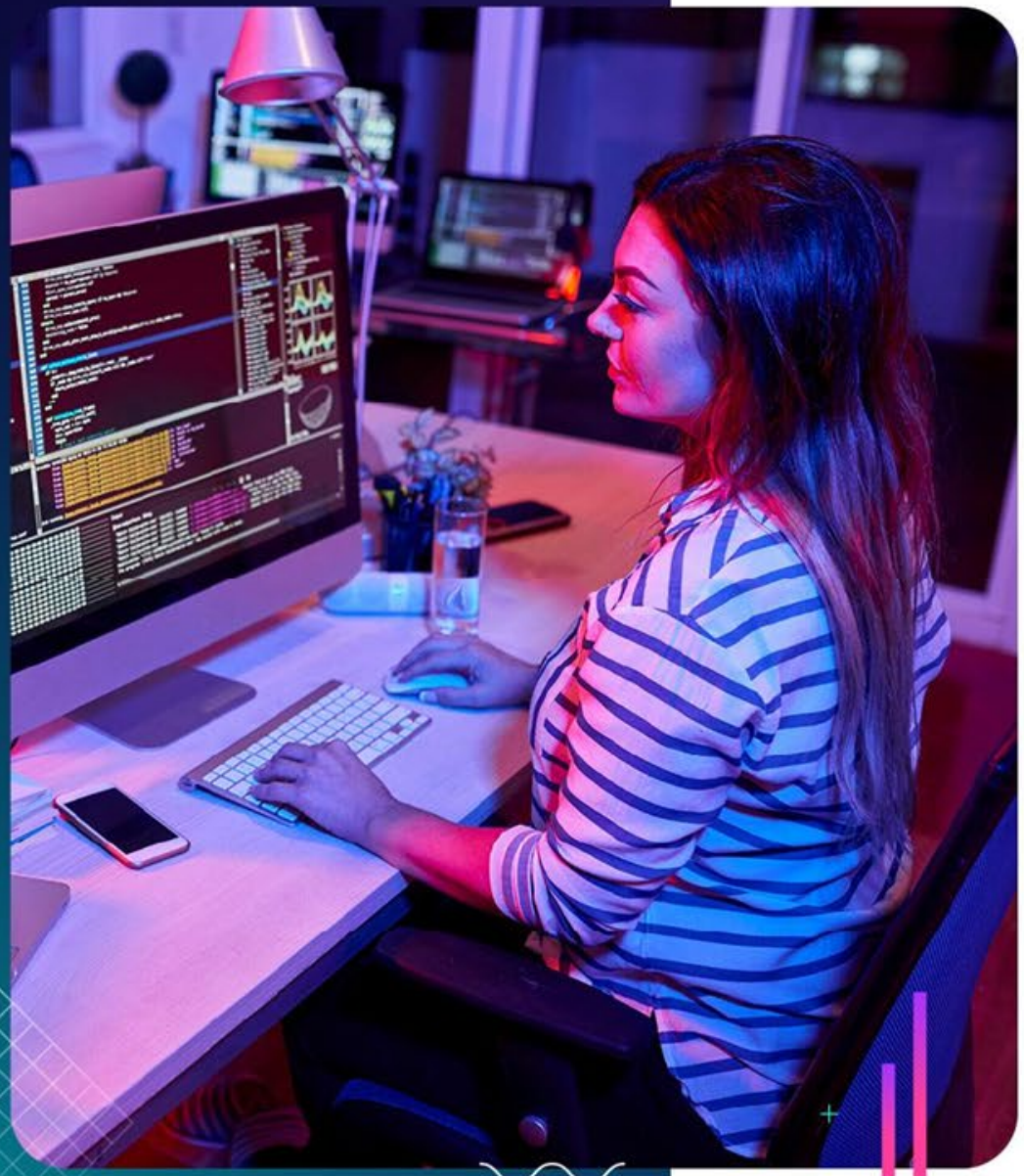
Strings:

- Binarios codificados en UTF-8
- Secuencia de caracteres Unicode, normalmente escritos entre cadenas de comillas dobles (“Hola Mundo”)
- Concatenación
- Interpolación

```
iex> "hello" <> " " <> "world"
"hello world"
```

```
iex> name = "joe"
iex> "hello #{name}"
"hello joe"
```





Vamos a ver algunas funciones

- String.length/1
- String.split/2
- String.contains?/2
- String.downcase/1
- String.duplicate/2
- String.upcase/1
- String.ends_with?/2
- String.first/1
- String.last/1
- String.match?/2
- String.replace/3
- String.reverse/1
- String.starts_with?/2
- String.trim/2
- String.to_integer/2
- String.to_float/2
- String.ends_with?/2
- String.at/2

Conclusiones

- Elixir es un lenguaje de programación moderno construido sobre la máquina virtual Erlang. Hereda muchas de las características de Erlang pero Elixir ofrece una sintaxis más moderna, herramientas más amplias y una comunidad de desarrolladores cada vez mayor.
- Elixir se adapta bien a diversos casos de uso, como la programación web, el desarrollo backend, los sistemas distribuidos y la comunicación en tiempo real.
- Está diseñado para ser altamente escalable y tolerante a fallos, por lo que es una opción popular para la construcción de sistemas distribuidos y aplicaciones web.
- Elixir es un lenguaje potente y flexible que proporciona una base sólida para crear sistemas distribuidos, de alto rendimiento y tolerantes a fallos.

[Mishell Yagual Mendoza]
[mishell.yagual@sofka.com.co]
Technical Coach
Sofka U

Preguntas & Respuestas

#ElDesafíoEsContigo

Calle 12 # 30-80 Medellín

Calle 85 # 11 – 53 Int 6 Of. 301 Bogotá

+57 604 266 4547

info@sofka.com.co

www.sofka.com.co

Síguenos

Sofka Technologies

Sofka_Technologies

Elixir

Programación más funcional que nunca

Semana 1 – MC #2

SofkaU

#ElDesafíoEsContigo

Temas

01 Flujos de control y guardas

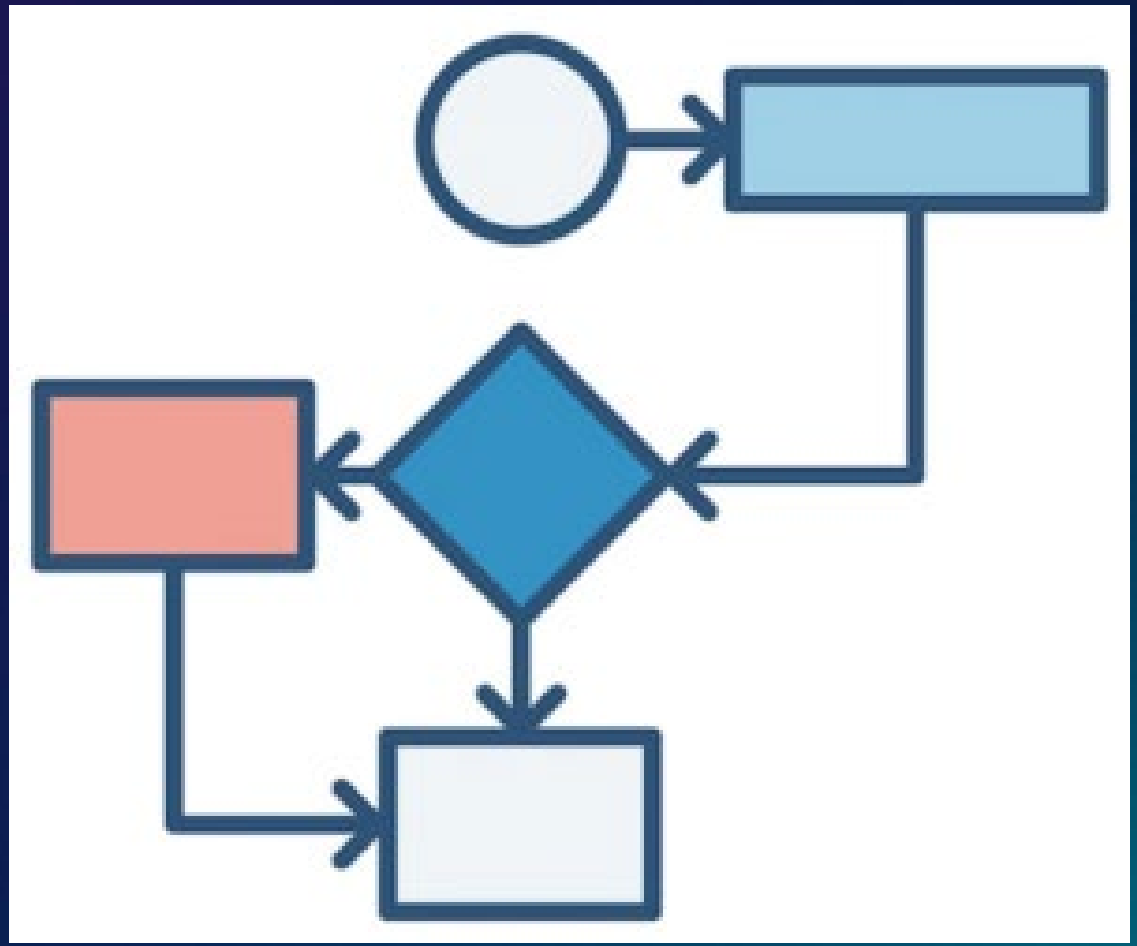
02 Colecciones I

03 Correspondencia de patrones

04 Operadores(^, _)

Elixir

Flujos de control

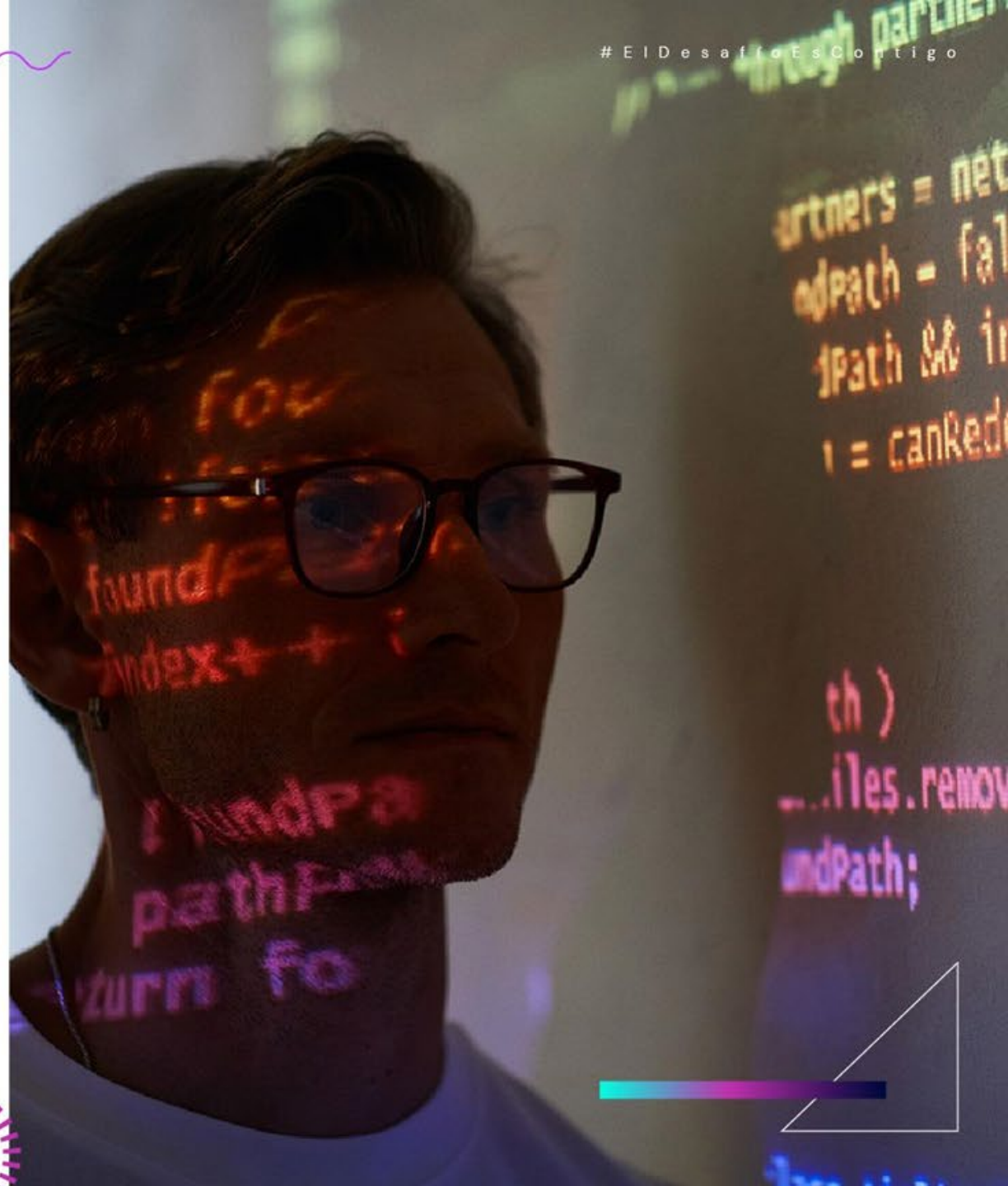


with

- Proporciona una sintaxis concisa para manejar una serie de coincidencias de forma que se lea como una sentencia.

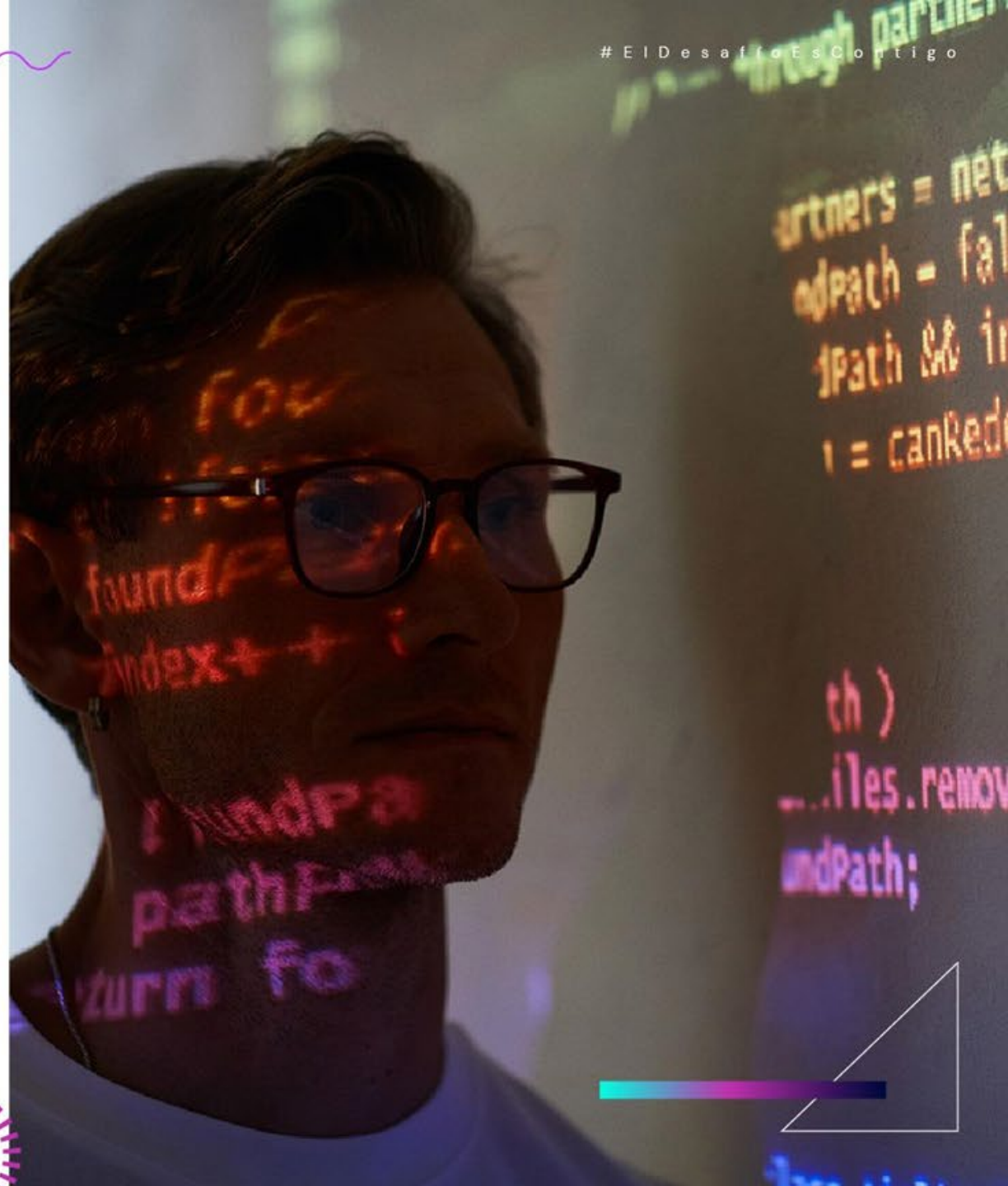
- Toma una o más expresiones que devuelven una tupla {:ok, value} o un átomo :error.

```
with {:ok, result1} <- do_something(arg1),
     {:ok, result2} <- do_something_else(result1, arg2),
     {:ok, result3} <- do_one_more_thing(result2) do
  {:ok, final_result}
else
  :error -> :error
end
```



if - else

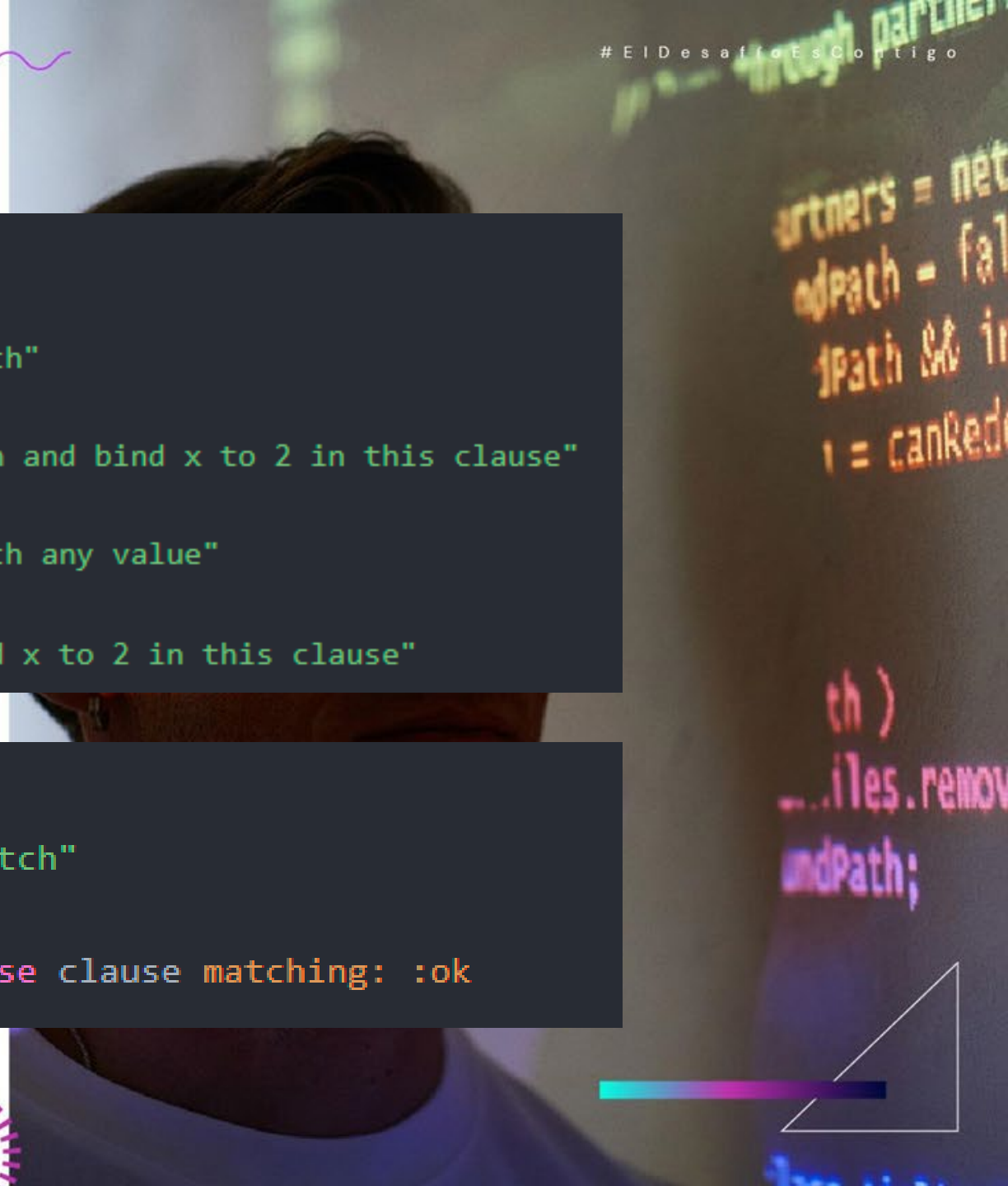
```
iex> if String.valid?("Hello") do
...>   "Valid string!"
...> else
...>   "Invalid string."
...> end
"Valid string!"

iex> if "a string value" do
...>   "Truthy"
...> end
"Truthy"
```



case

```
iex> case {1, 2, 3} do
...>   {4, 5, 6} ->
...>     "This clause won't match"
...>   {1, x, 3} ->
...>     "This clause will match and bind x to 2 in this clause"
...>   _ ->
...>     "This clause would match any value"
...> end
"This clause will match and bind x to 2 in this clause"
```

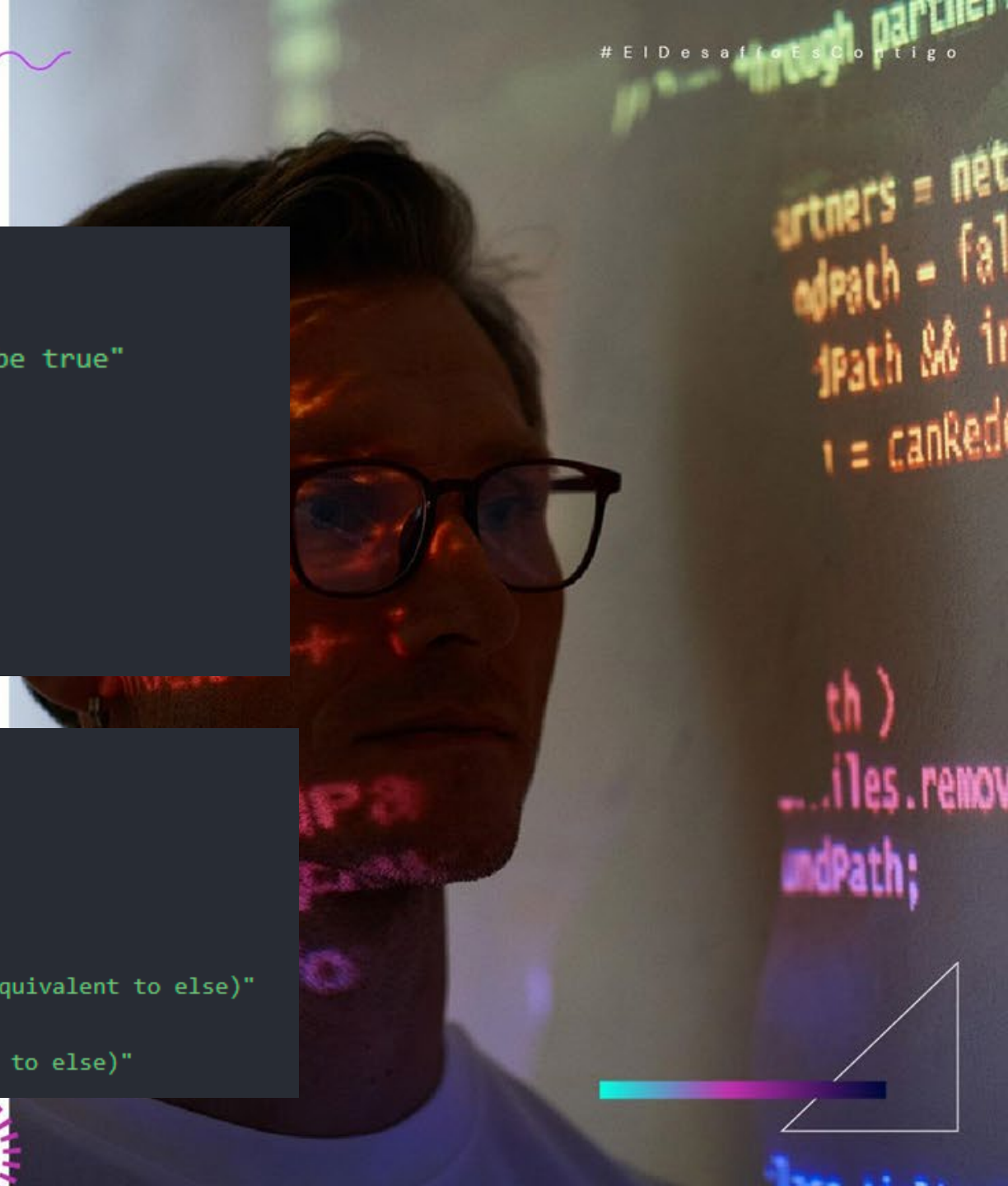
```
iex> case :ok do
...>   :error -> "Won't match"
...> end
** (CaseClauseError) no case clause matching: :ok
```



cond

```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This will not be true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   1 + 1 == 2 ->
...>     "But this will"
...> end
"But this will"
```

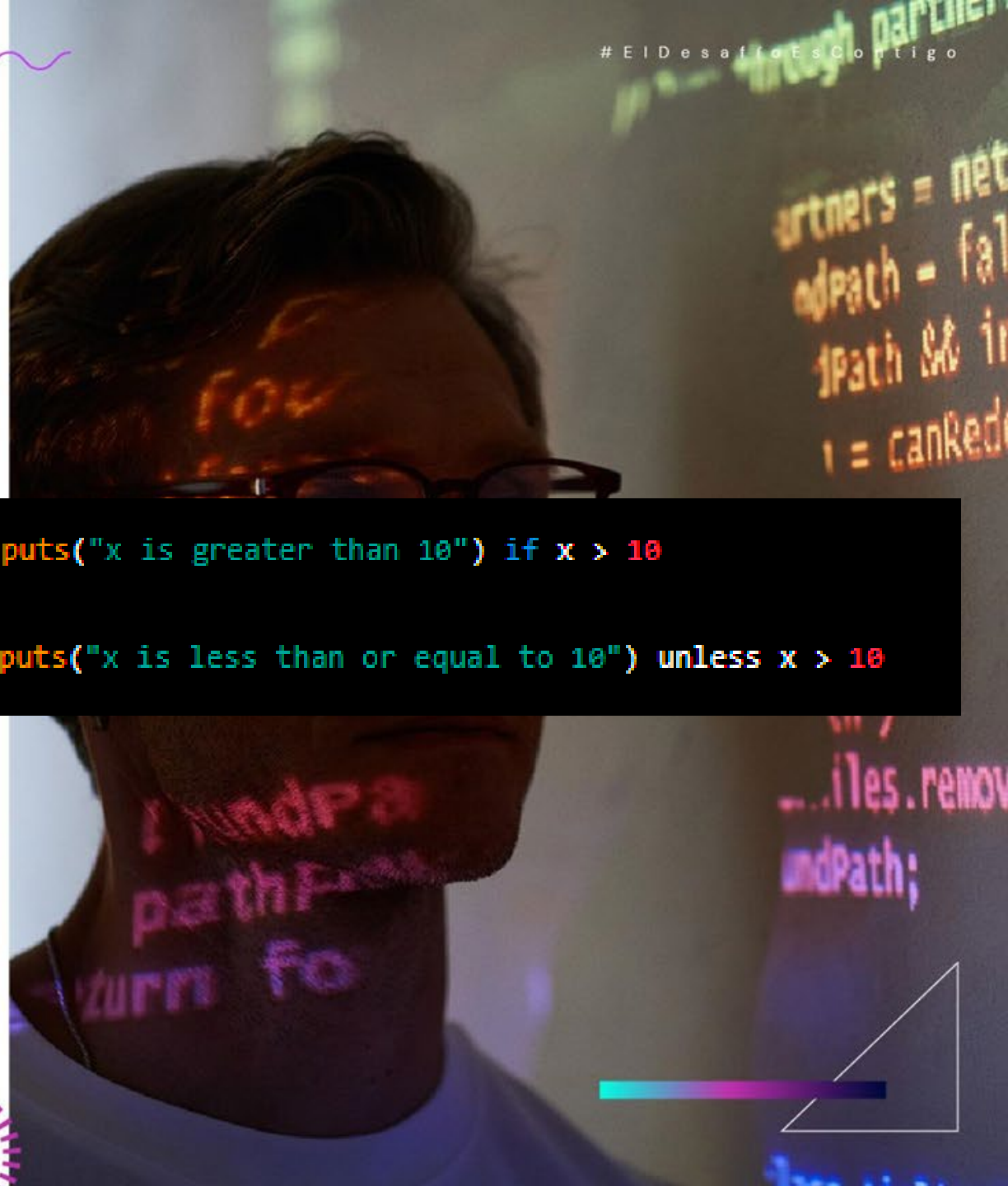
```
iex> cond do
...>   2 + 2 == 5 ->
...>     "This is never true"
...>   2 * 2 == 3 ->
...>     "Nor this"
...>   true ->
...>     "This is always true (equivalent to else)"
...> end
"This is always true (equivalent to else)"
```



If/unless

```
iex> if "a string value" do
...>   "Truthy"
...> end
"Truthy"
```

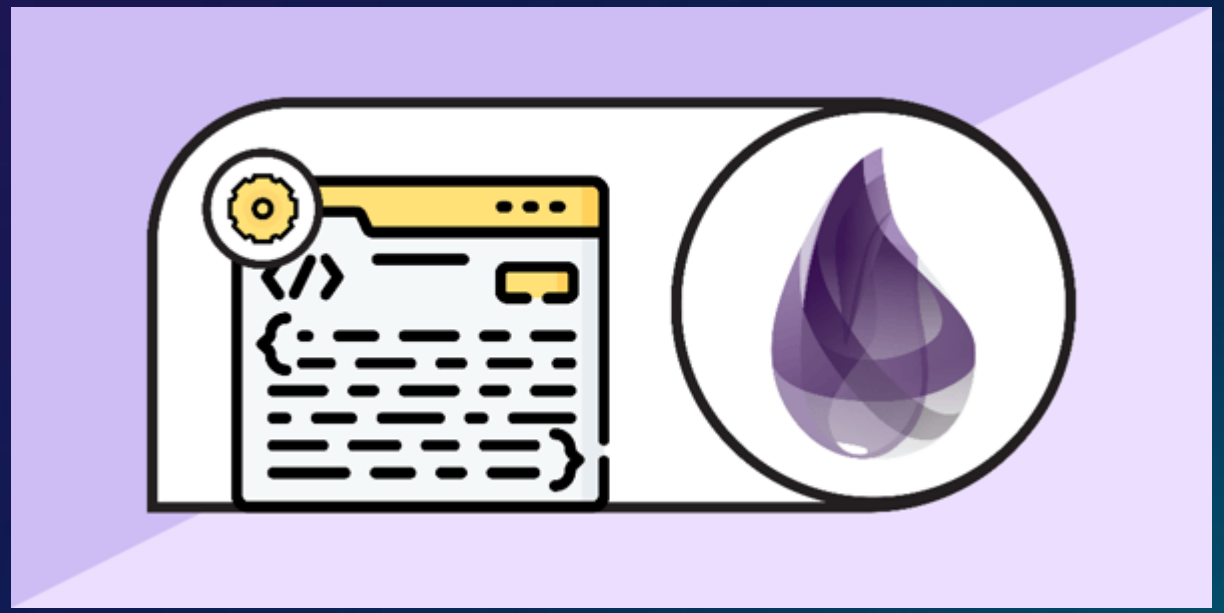
```
iex> unless is_integer("hello") do
...>   "Not an Int"
...> end
"Not an Int"
```




```
IO.puts("x is greater than 10") if x > 10

IO.puts("x is less than or equal to 10") unless x > 10
```

Elixir

Colecciones



Listas:

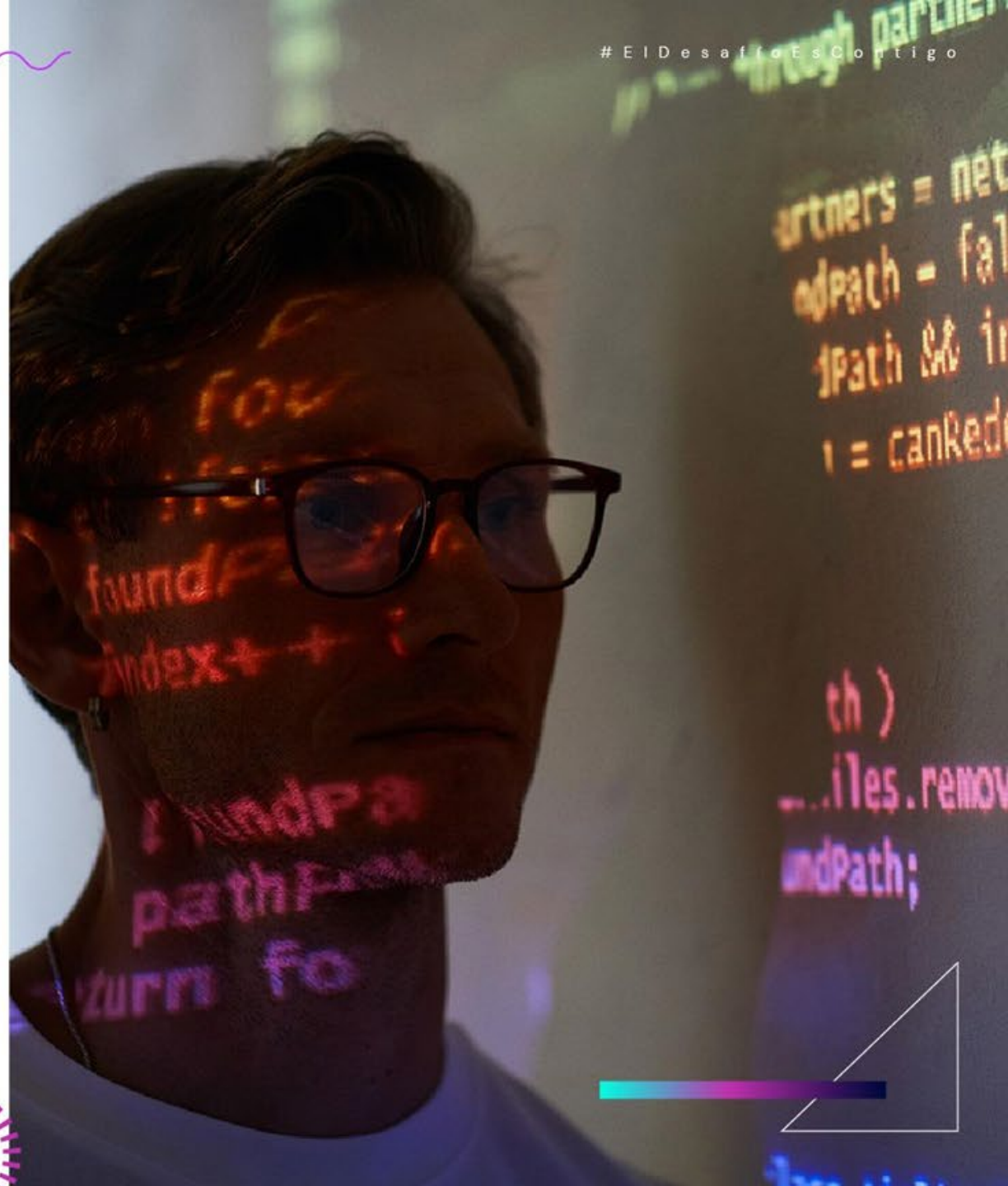
- Conocidas también como Linked list.

```
iex> [head | tail] = [1, 2, 3]
iex> head
1
iex> tail
[2, 3]
```

- Se almacenan en distintas ubicaciones de la memoria y que se siguen mediante el uso de referencias.

- **Inmutables**

```
iex> [1, "two", 3, :four]
[1, "two", 3, :four]
```



Listas: “Modificación”

- Más ‘barato’ que una tupla, aunque depende.
- Añadir al principio de una lista es una operación O(1), mientras que añadir al final de una lista es una operación O(n).

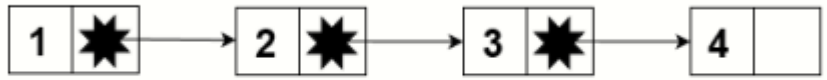


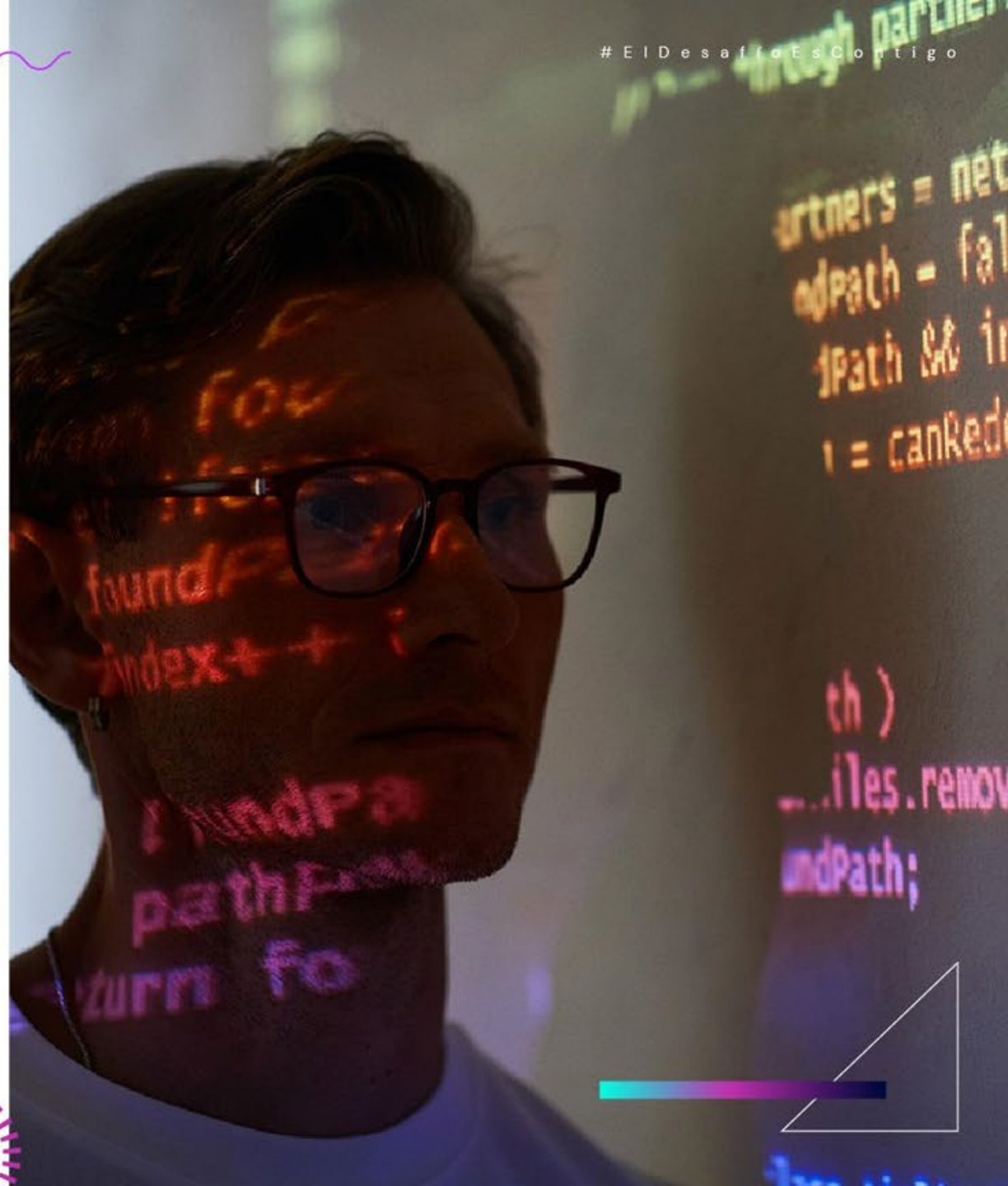
Diagram of a Linked List

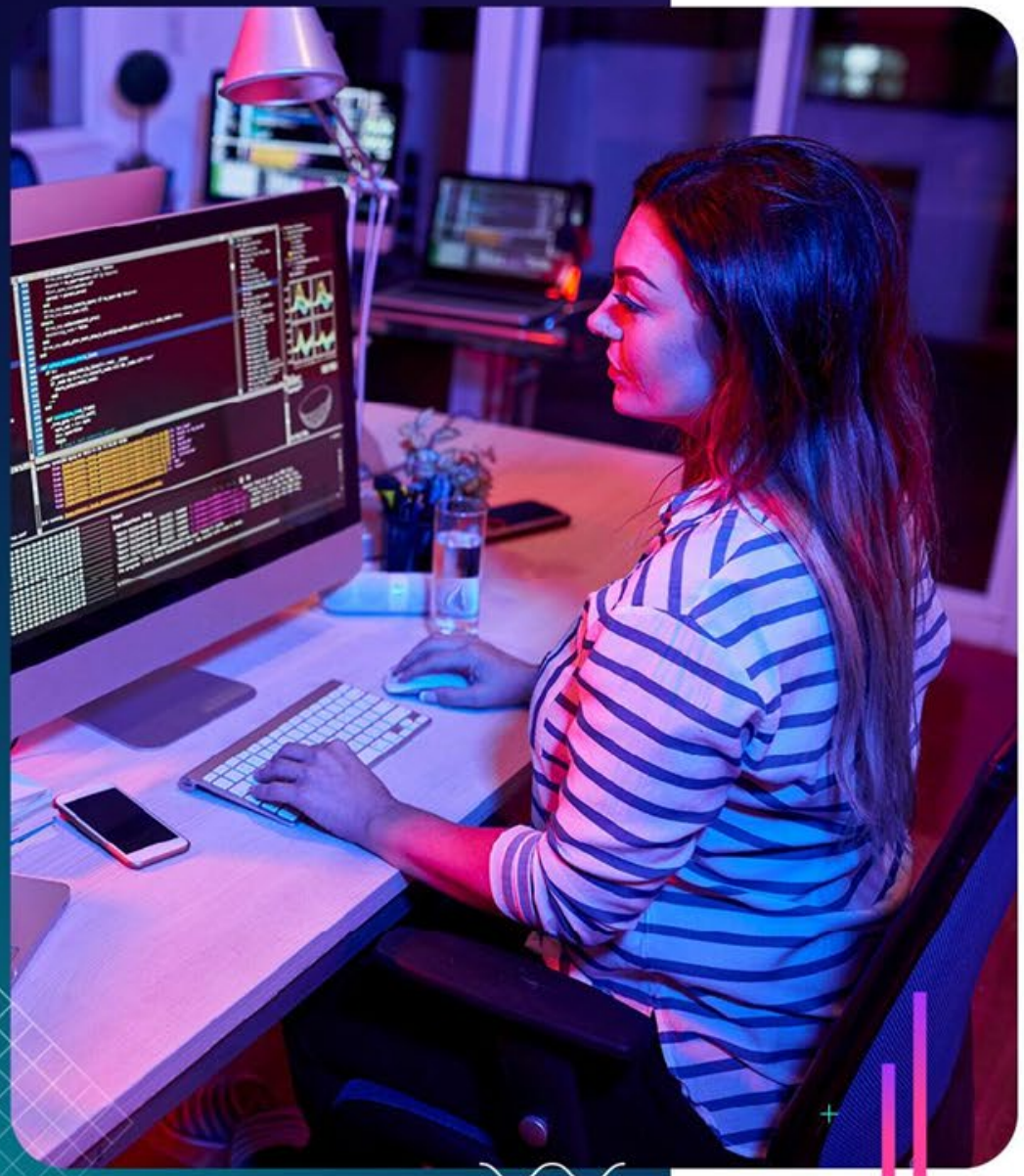
A new node is prepended to the linked list

```
iex(3)> values
[1, "Hello", 2.34, :list]
iex(4)> values
[1, "Hello", 2.34, :list]
iex(5)> ["new" | values]
["new", 1, "Hello", 2.34, :list]
iex(6)> values
[1, "Hello", 2.34, :list]
iex(7)> values = ["new" | values]
["new", 1, "Hello", 2.34, :list]
iex(8)> values
["new", 1, "Hello", 2.34, :list]
iex(9)>
```

Listas: “Modificación”

```
iex(3)> values
[1, "Hello", 2.34, :list]
iex(4)> values
[1, "Hello", 2.34, :list]
iex(5)> ["new" | values]
["new", 1, "Hello", 2.34, :list]
iex(6)> values
[1, "Hello", 2.34, :list]
iex(7)> values = ["new" | values]
["new", 1, "Hello", 2.34, :list]
iex(8)> values
["new", 1, "Hello", 2.34, :list]
iex(9)>
```





Vamos a ver algunas funciones

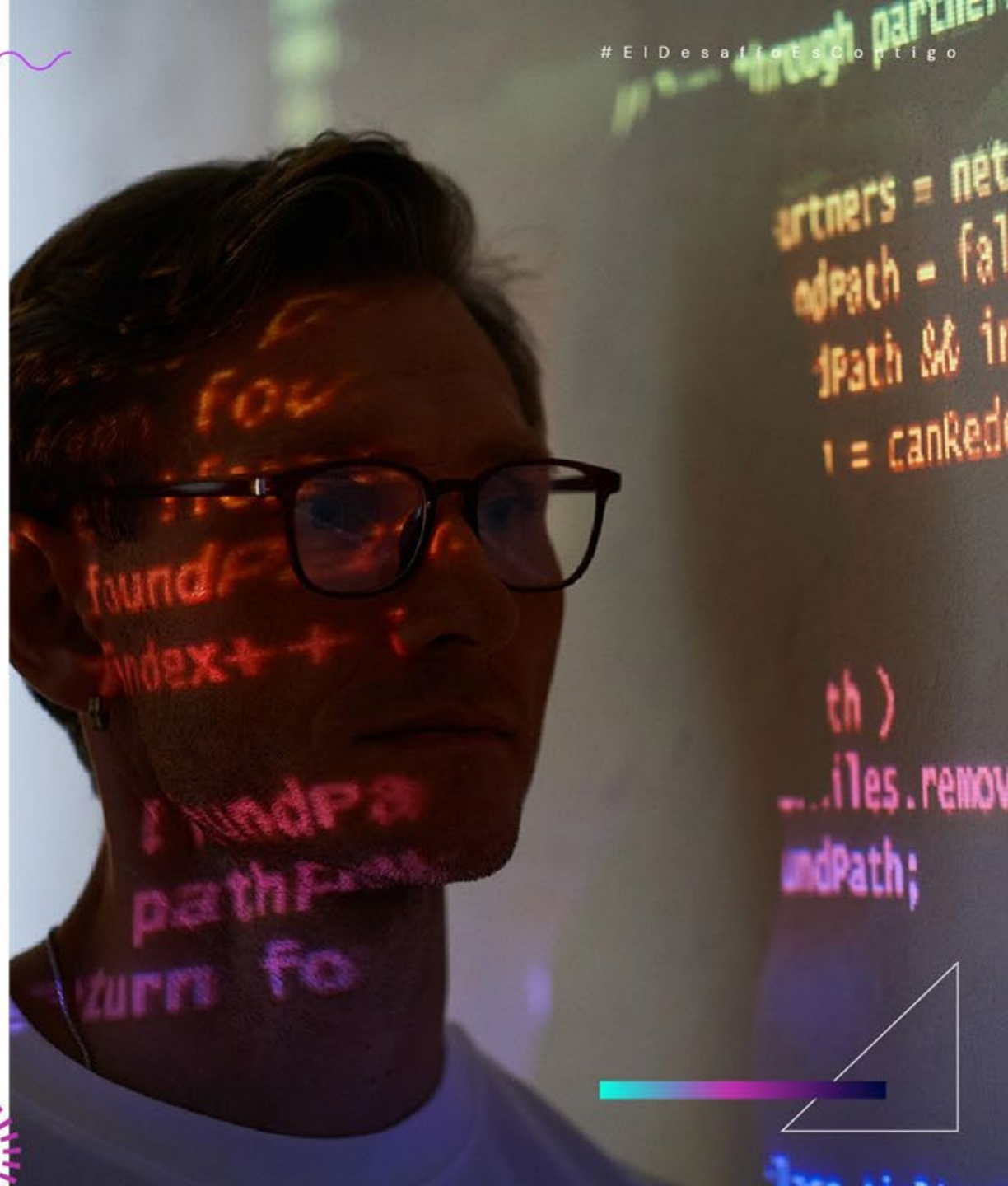
- Las listas implementan el protocolo Enumerable, que permite el uso de las funciones del módulo Enum.

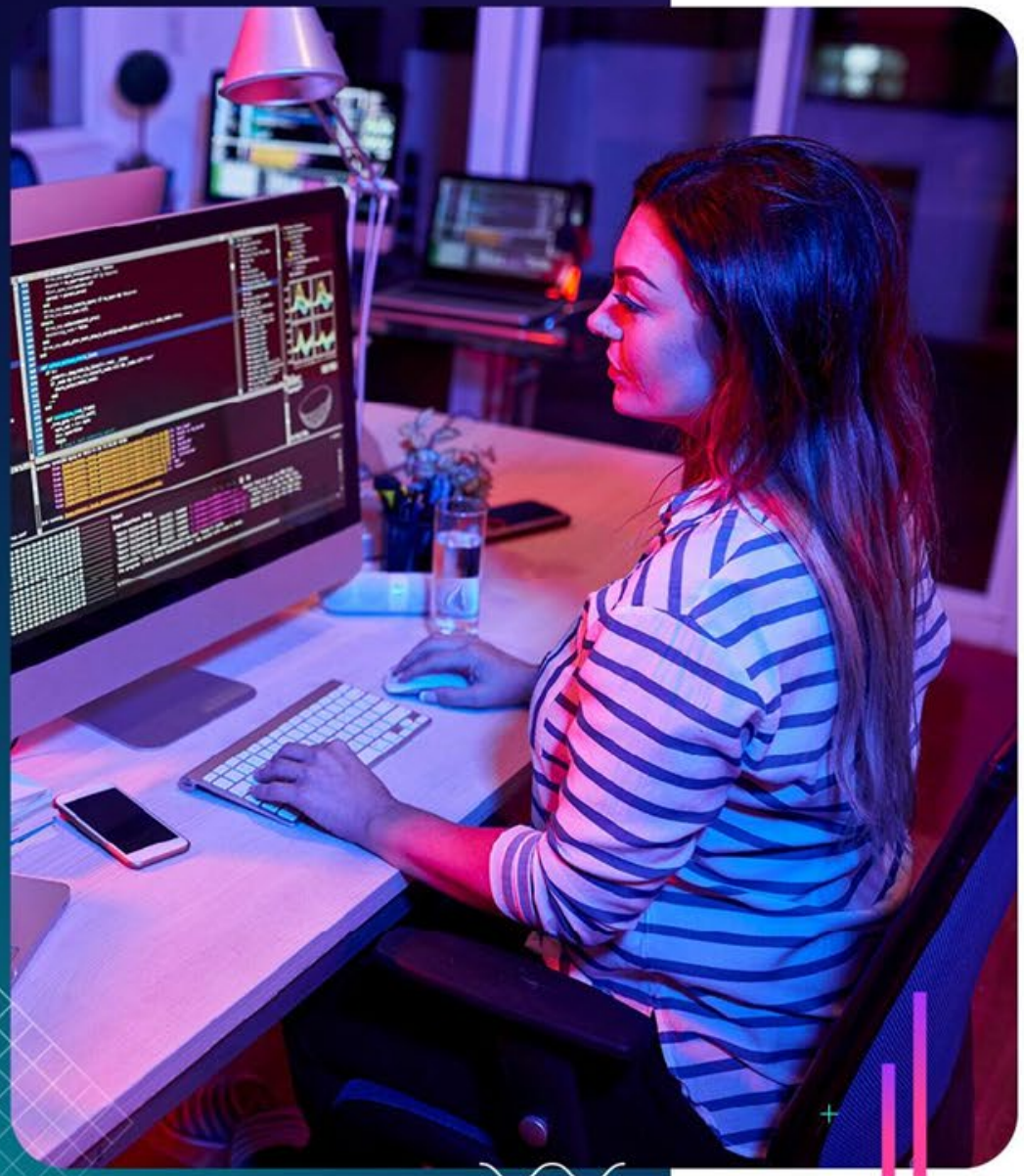
- List.delete/2
- List.delete_at/2
- List.first/2
- List.insert_at/3
- List.last/2
- List.replace_at/2
- List.update_at/3
- List.to_string/1

Enum:

- Proporciona un conjunto de algoritmos para trabajar con enumerables.
- En Elixir, un enumerable es cualquier tipo de datos que implementa el protocolo Enumerable.
- Actúa tan pronto sus funciones sean invocadas, teniendo un comportamiento ‘eager’.

```
iex> Enum.map([1, 2, 3], fn x -> x * 2 end)
[2, 4, 6]

iex> Enum.sum([1, 2, 3])
6
```





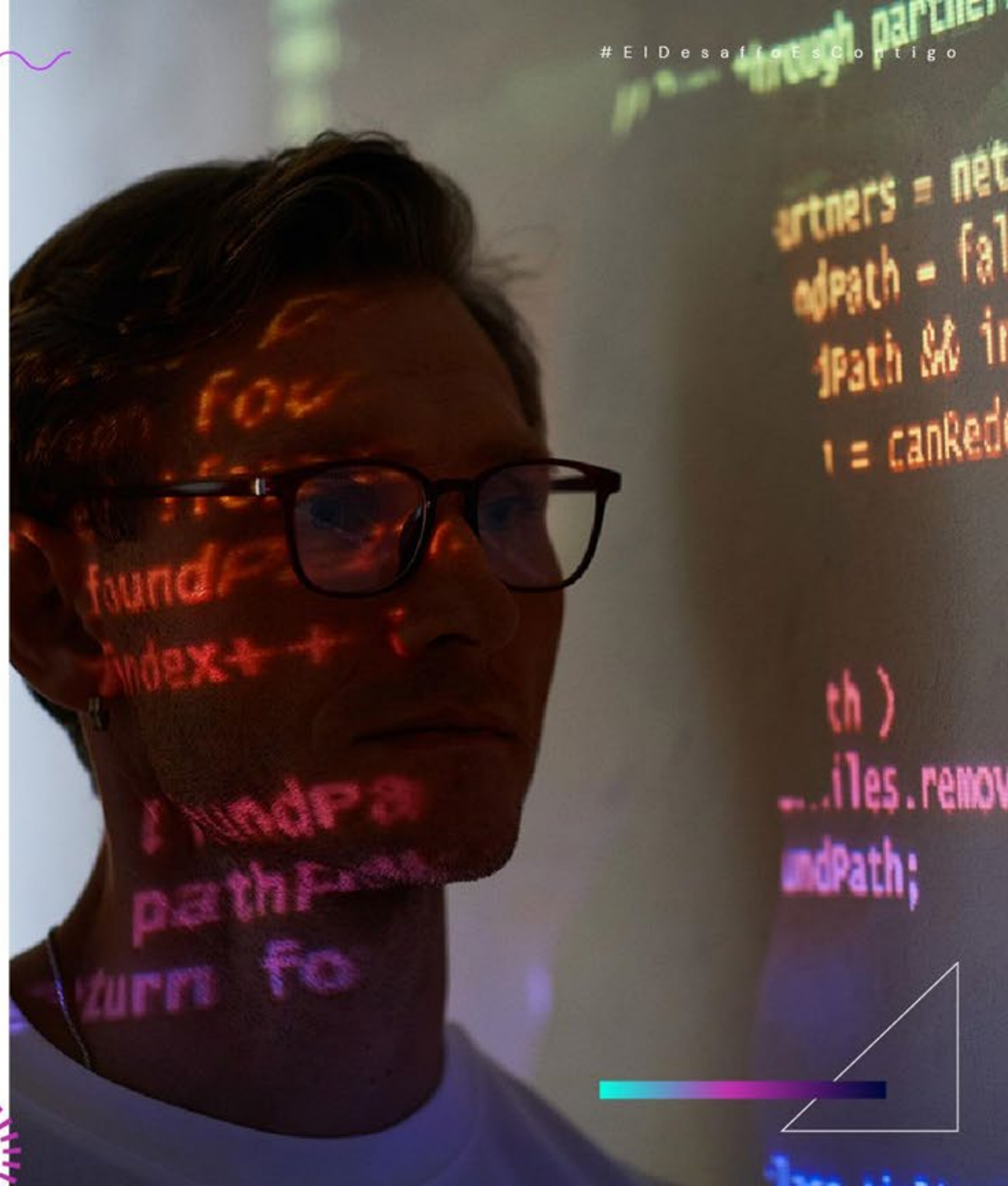
Vamos a ver algunas funciones

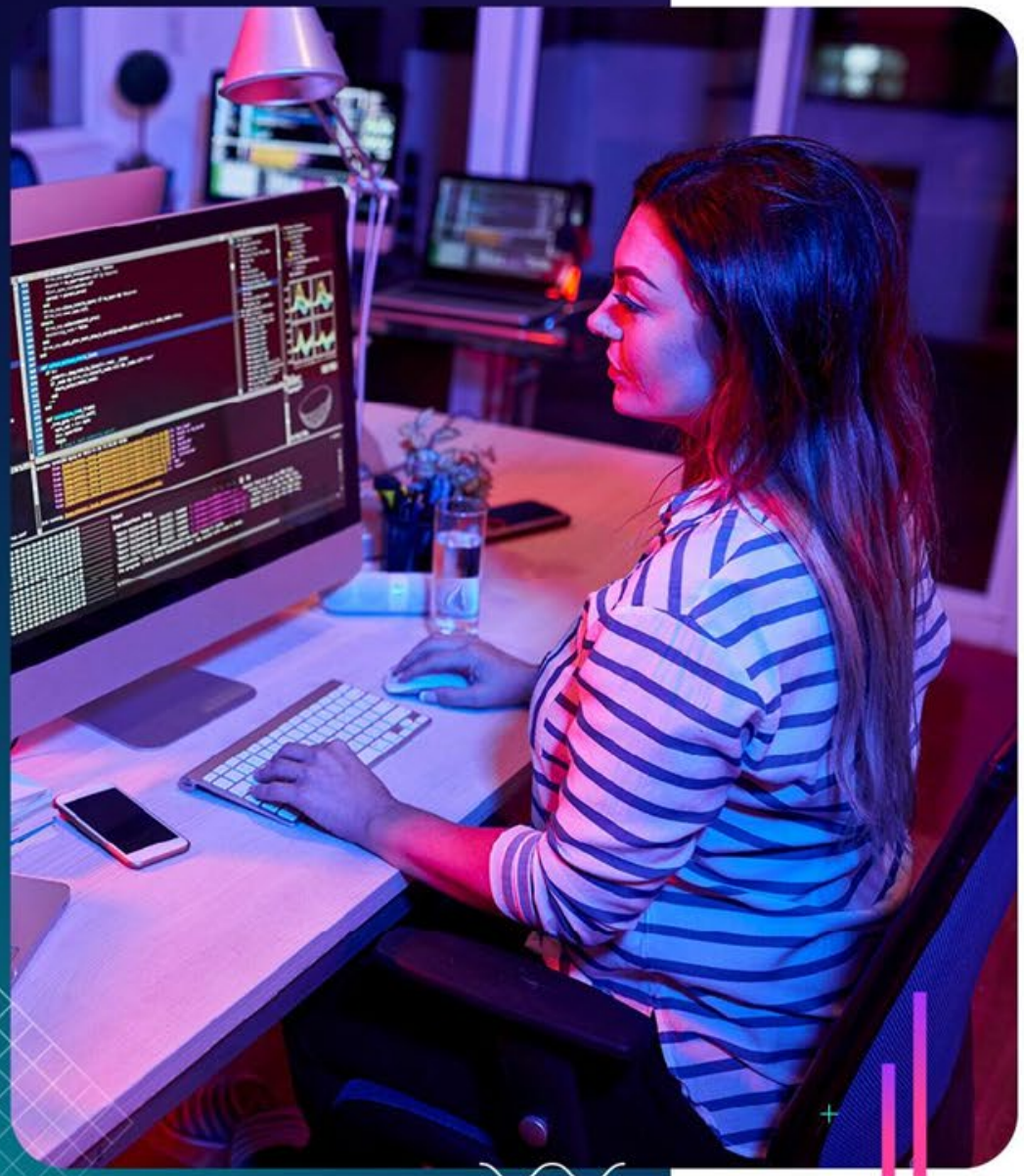
- Enum.filter/2
- Enum.at/2
- Enum.take/2
- Enum.reduce/3
- Enum.sort/1-2
- Enum.map/2
- Enum.reverse/1
- Enum.sum/1
- Enum.find/2
- Enum.join/1-2
- Enum.max/1-2
- Enum.min/1-2
- Enum.product/1
- Enum.random/1
- Enum.reject/2

Map:

- La estructura de datos clave-valor más utilizada en Elixir.
- Un mapa puede ser creado con la estructura %{ }

```
%{
  "Monday" => 28,
  "Tuesday" => 29,
  "Wednesday" => 29,
  "Thursday" => 24,
  "Friday" => 16,
  "Saturday" => 16,
  "Sunday" => 20
}
```





Características

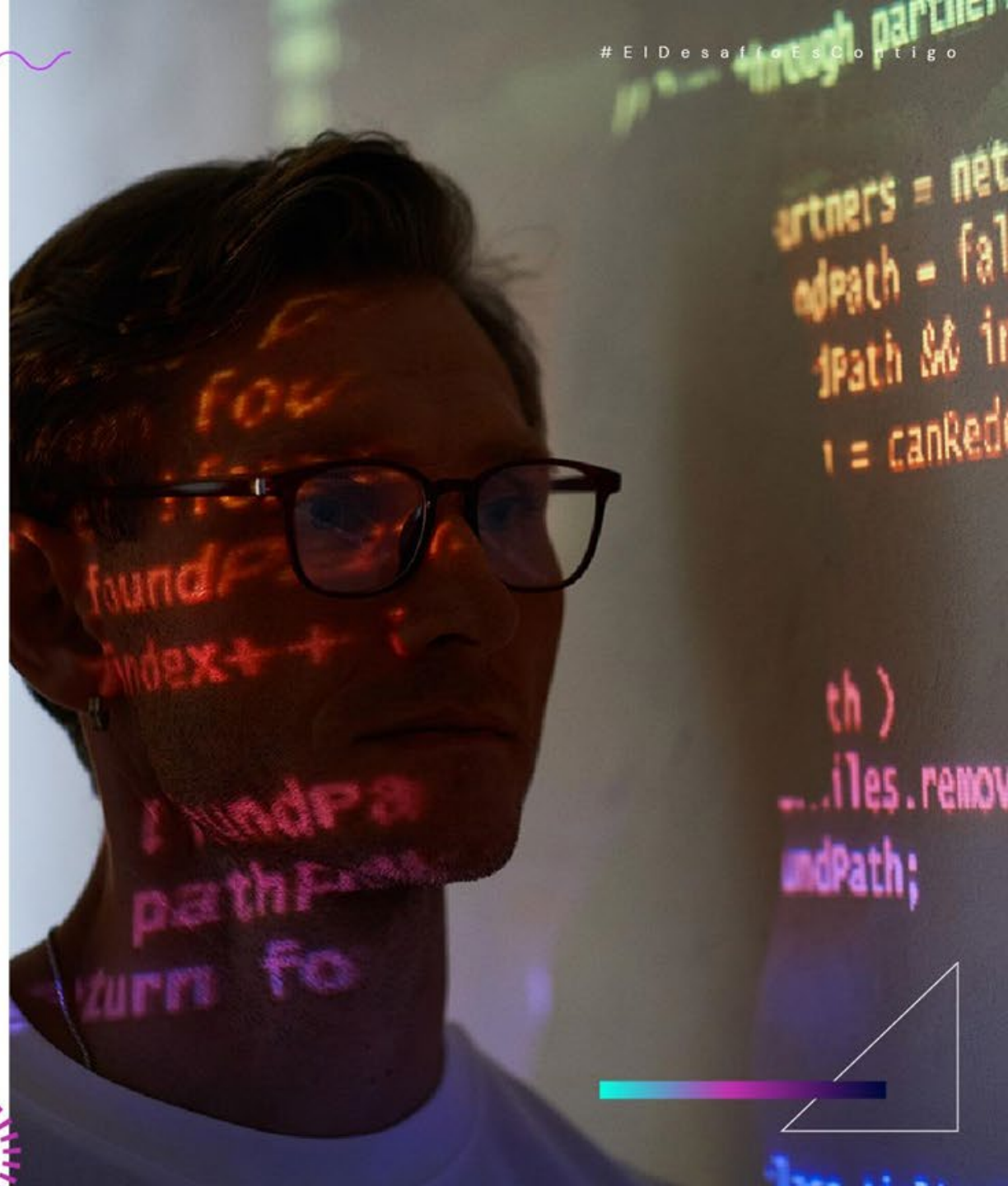
- Los pares clave-valor en un mapa no siguen ningún orden.
- Los mapas no imponen ninguna restricción al tipo de clave: cualquier cosa puede ser una clave en un mapa.
- Los mapas no permiten claves duplicadas.
- Cuando la clave de un par clave-valor es un átomo, se puede utilizar la sintaxis abreviada clave: valor

```
iex> %{a: 1, b: 2}
%{a: 1, b: 2}
```

Pattern matching

- La concordancia de patrones es una parte poderosa de Elixir.
- Nos permite hacer coincidir valores simples, estructuras de datos e incluso funciones.

```
iex(12)> [a, b, c] = [1, "Holi", :ok]
[1, "Holi", :ok]
iex(13)> a
1
iex(14)> b
"Holi"
iex(15)> c
:ok
iex(16)> %{weight: w} = %{weight: 50}
%{weight: 50}
iex(17)> w
50
iex(18)> {status, code} = {:error, 404}
{:error, 404}
iex(19)> status
:error
iex(20)> code
404
```

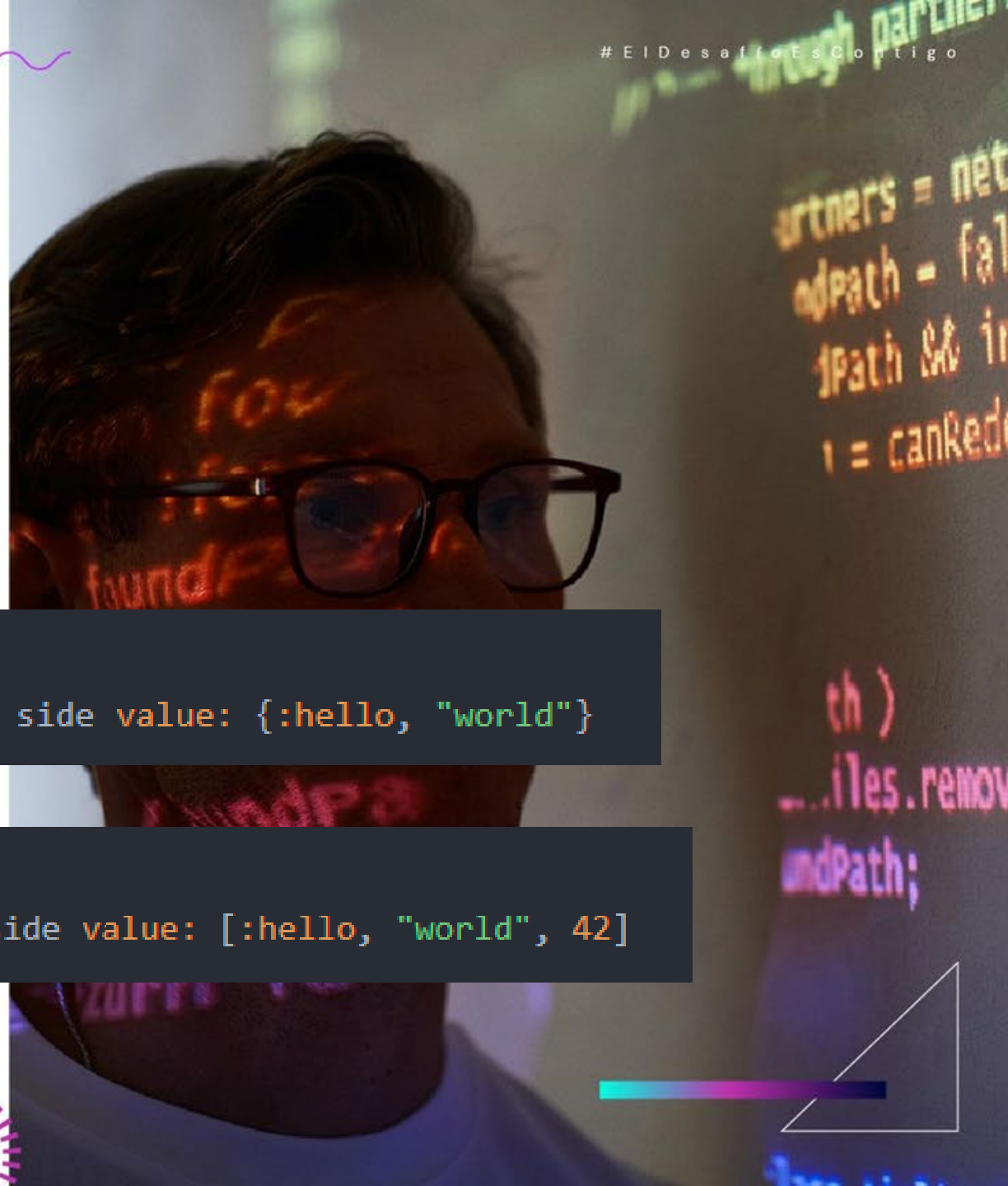


Pattern matching

```
iex> {a, b, c} = {:hello, "world", 42}
{:hello, "world", 42}
iex> a
:hello
iex> b
"world"
```

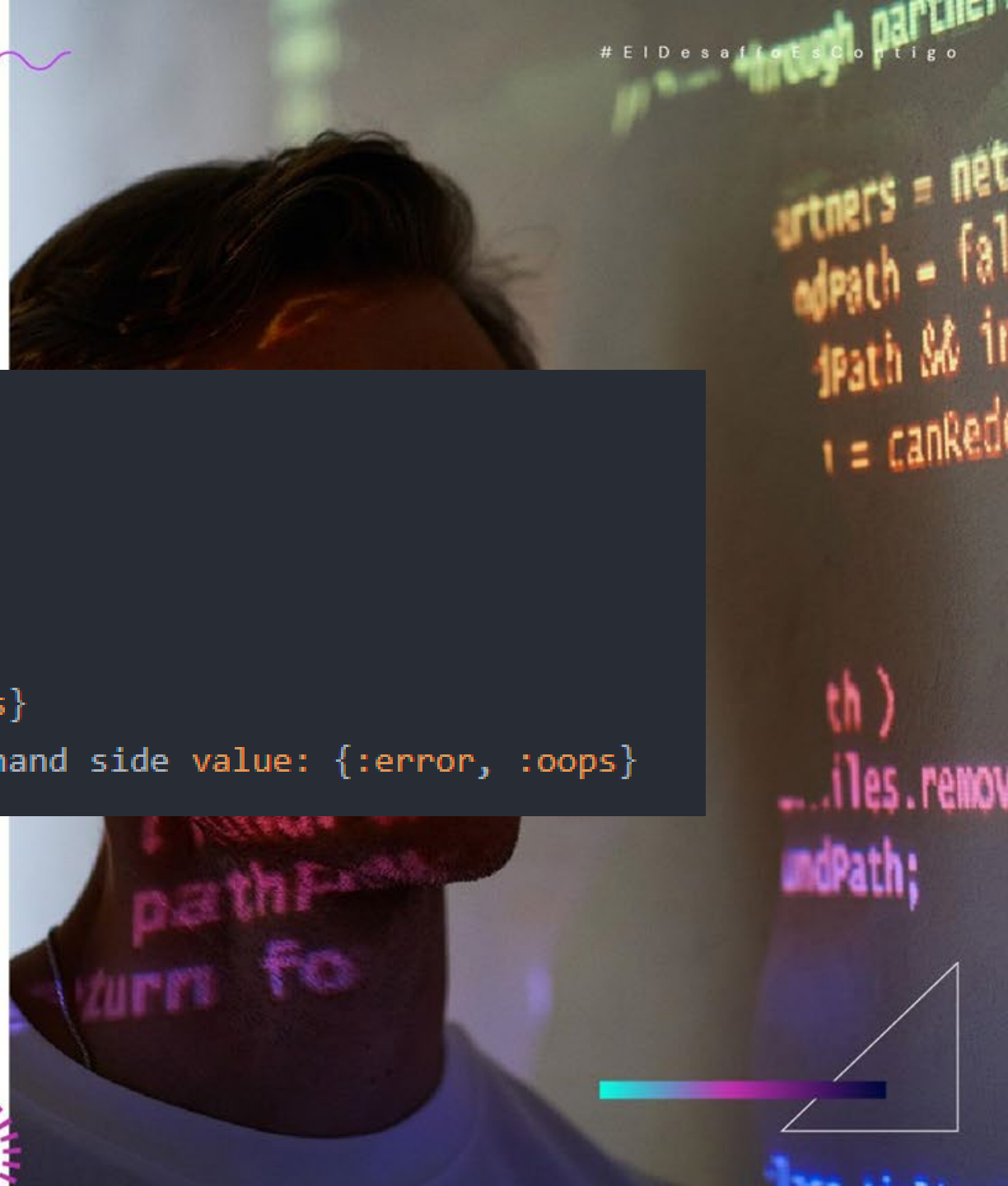
```
iex> {a, b, c} = {:hello, "world"}
** (MatchError) no match of right hand side value: {:hello, "world"}
```

```
iex> {a, b, c} = [:hello, "world", 42]
** (MatchError) no match of right hand side value: [:hello, "world", 42]
```



Pattern matching

```
iex> {:ok, result} = {:ok, 13}
{:ok, 13}
iex> result
13

iex> {:ok, result} = {:error, :oops}
** (MatchError) no match of right hand side value: {:error, :oops}
```

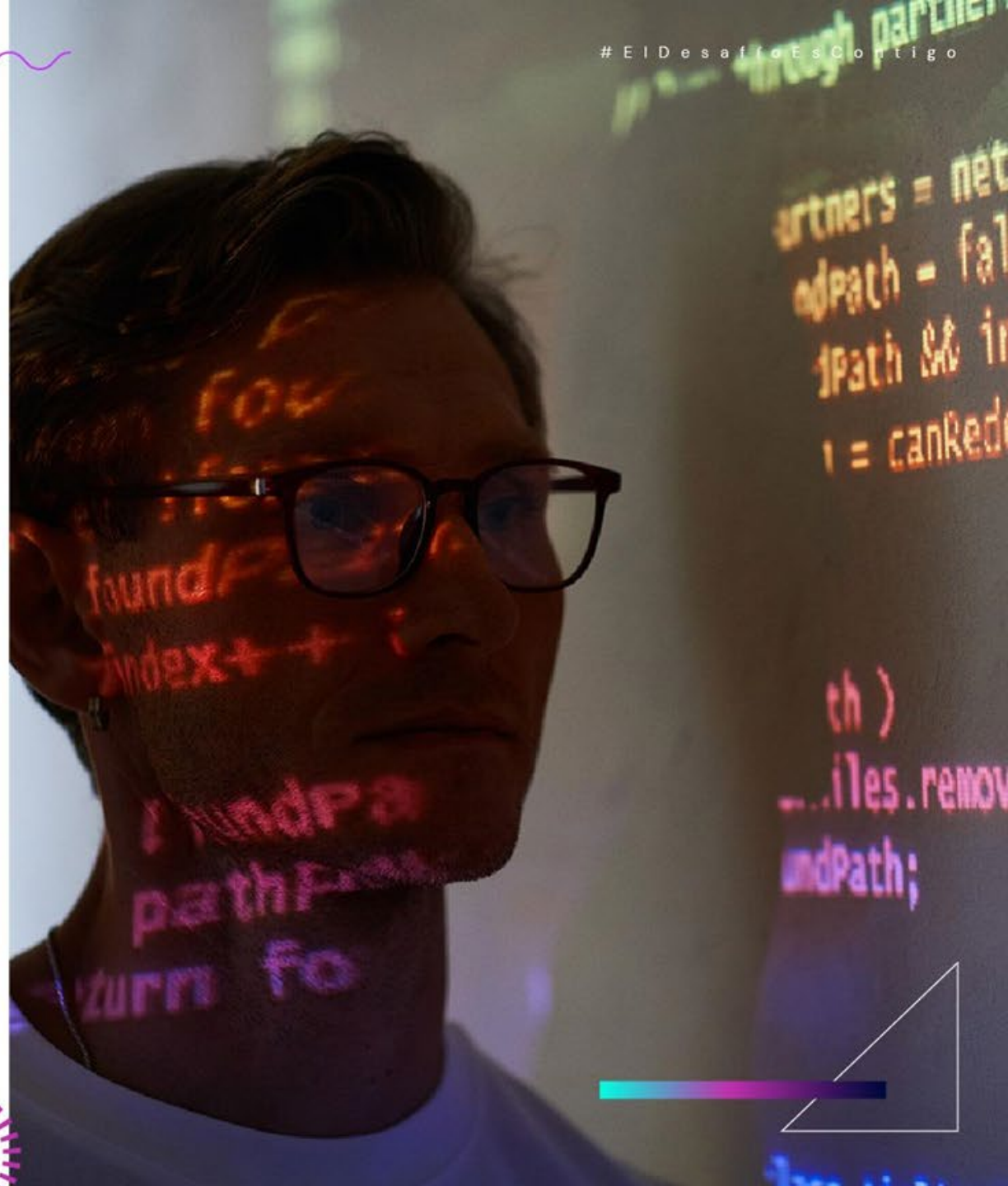


Guardas

- Expresión booleana que se evalúa en tiempo de ejecución para determinar si una cláusula concreta coincide con los argumentos que se han introducido. .

- Si la expresión de guarda es verdadera, la cláusula se considera coincidente (Pattern matching) y se ejecuta el código asociado a esa cláusula.

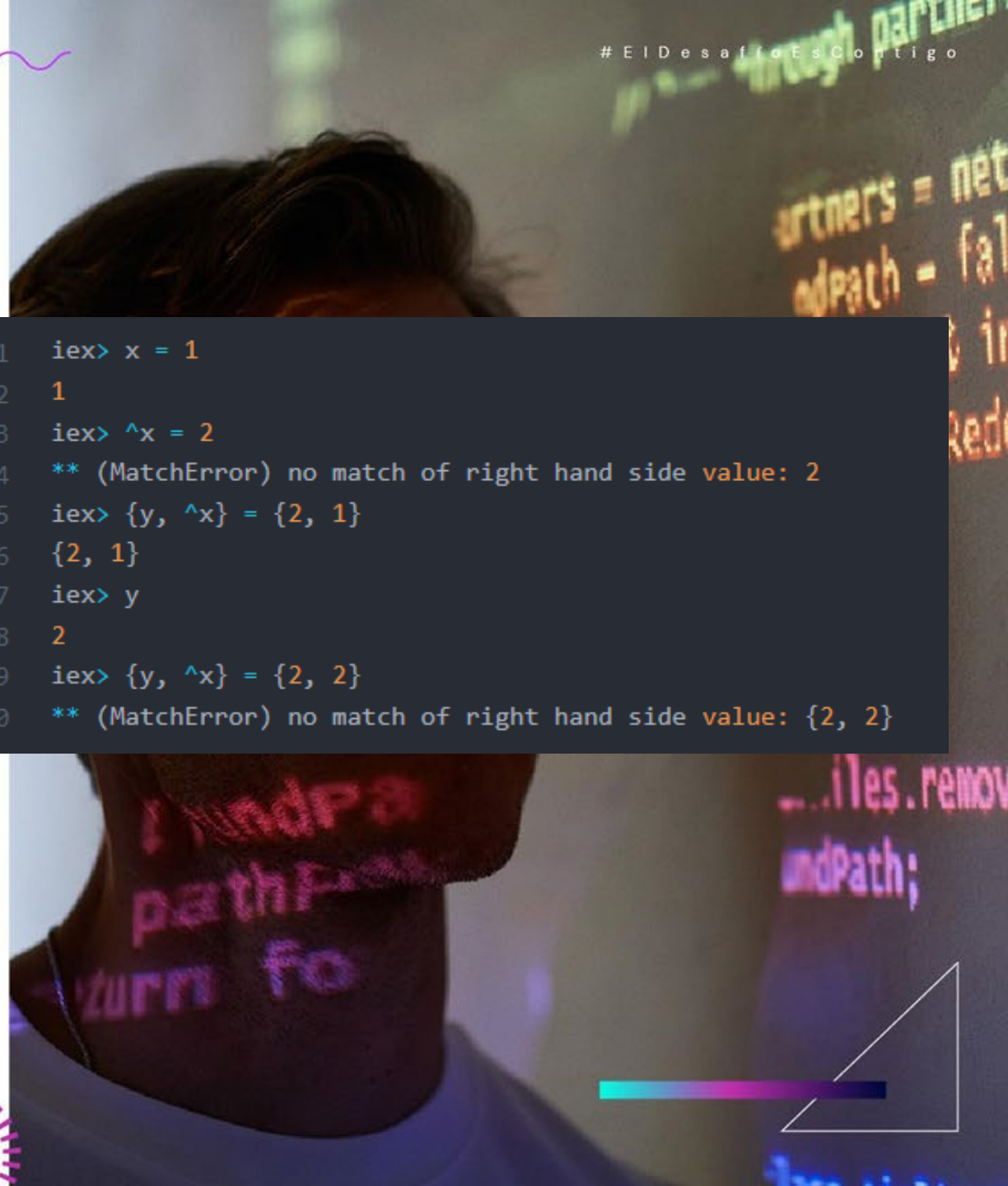
```
age = 25

case age do
  x when x < 18 -> IO.puts("You are a minor")
  x when x >= 18 and x < 65 -> IO.puts("You are an adult")
  _ -> IO.puts("You are a senior")
end
```



El operador pin (^)

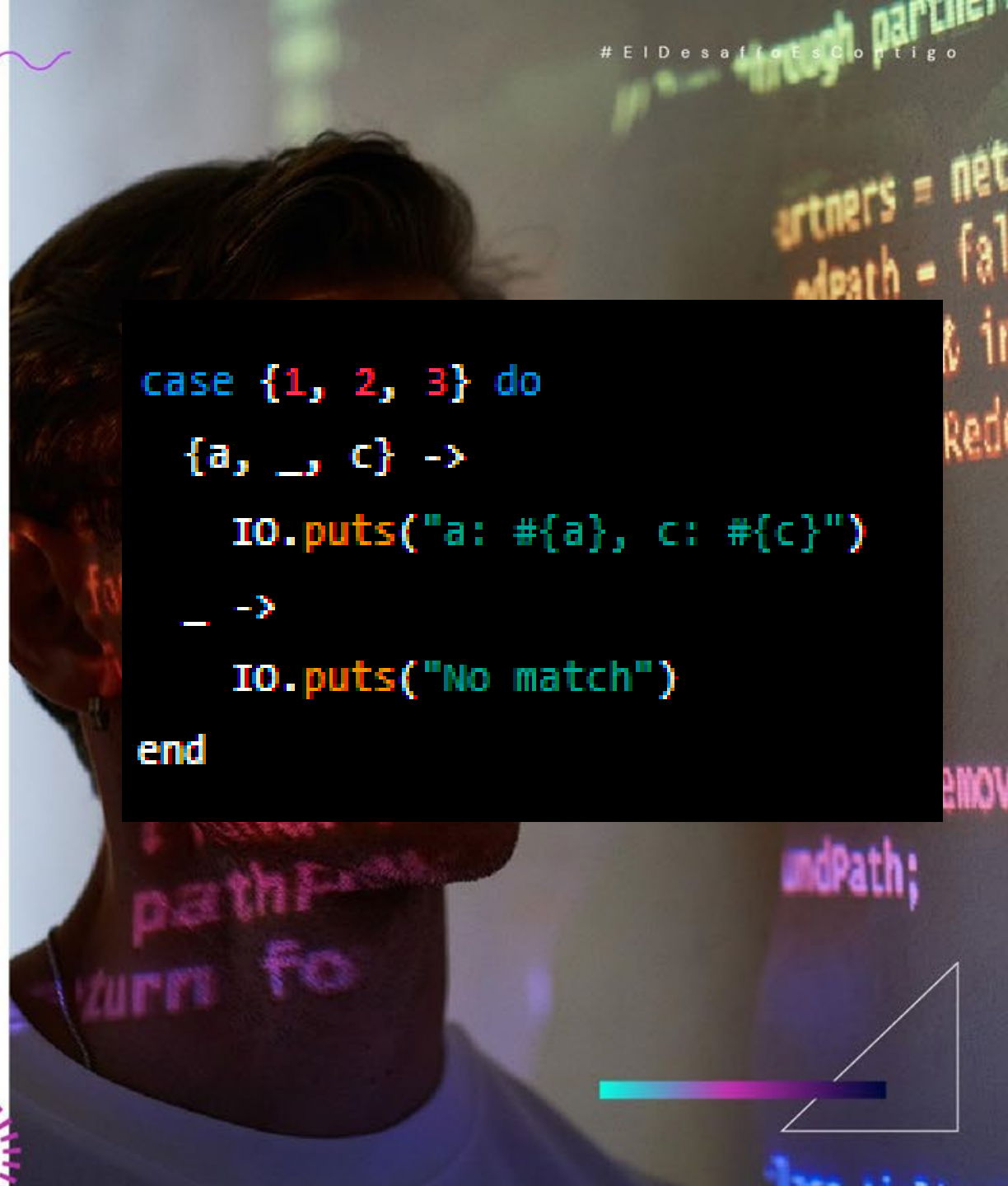
- Cuando se quiere comparar patrones con el valor de una variable existente en lugar de volver a vincular la variable.
- Se utiliza a menudo en las cabeceras de las funciones para especificar el valor esperado de una variable sin cambiar su valor

```
iex> x = 1
1
iex> ^x = 2
** (MatchError) no match of right hand side value: 2
iex> {y, ^x} = {2, 1}
{2, 1}
iex> y
2
iex> {y, ^x} = {2, 2}
** (MatchError) no match of right hand side value: {2, 2}
```



El operador de escape (_)

- Caracter comodín y se utiliza a menudo como marcador de posición para variables cuyos valores no son necesarios.
- Se puede utilizar para ignorar ciertas variables o para que coincida con cualquier valor.



Conclusiones

- El operador ^ se utiliza para hacer coincidir un patrón de un valor a una variable, indicando que la variable debe estar vinculada al valor si y sólo si el valor coincide con el patrón.
- Ambos operadores (^ , _) se utilizan con frecuencia en la concordancia de patrones, lo que permite concordar con valores específicos e ignorar otros.
- Las colecciones tienen una amplia variedad de usos, desde el almacenamiento de estructuras de datos simples hasta la creación de modelos de datos complejos.
- La concordancia de patrones se utiliza a menudo para trabajar con colecciones, lo que permite extraer y transformar datos de forma concisa y expresiva.

[Mishell Yagual Mendoza]
[mishell.yagual@sofka.com.co]
Technical Coach
Sofka U

Preguntas & Respuestas

#ElDesafíoEsContigo

Calle 12 # 30-80 Medellín

Calle 85 # 11 – 53 Int 6 Of. 301 Bogotá

+57 604 266 4547

info@sofka.com.co

www.sofka.com.co

Síguenos

Sofka Technologies

Sofka_Technologies