Elixir

Programación más funcional que nunca

Semana 2 – MC #3

SofkaU

#ElDesafíoEsContigo

Temas

01 Programación funcional

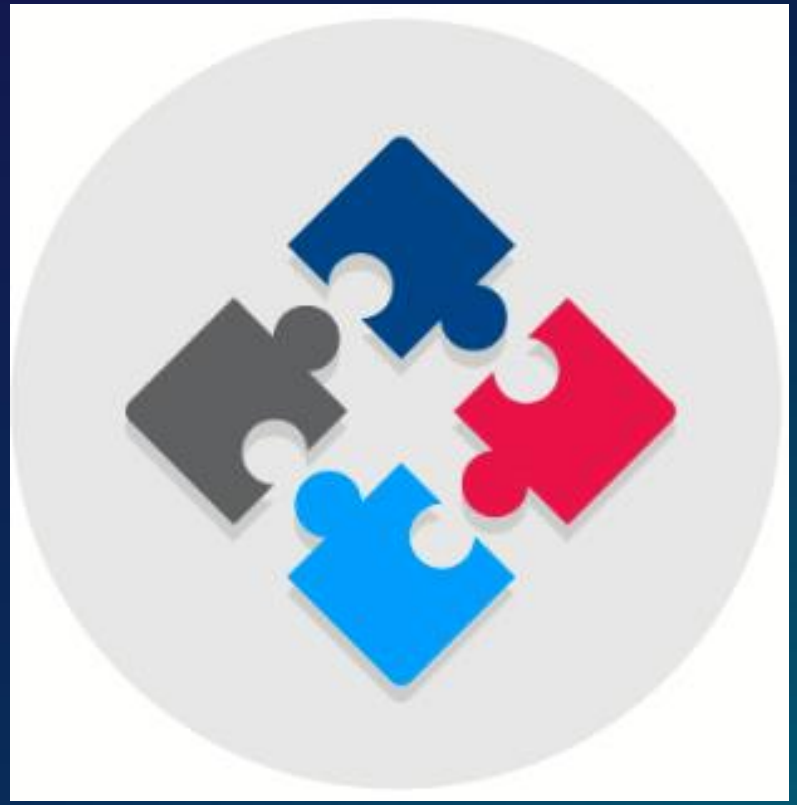
02 Funciones

03 Módulos, Alias e Import

04 Preguntas y respuestas

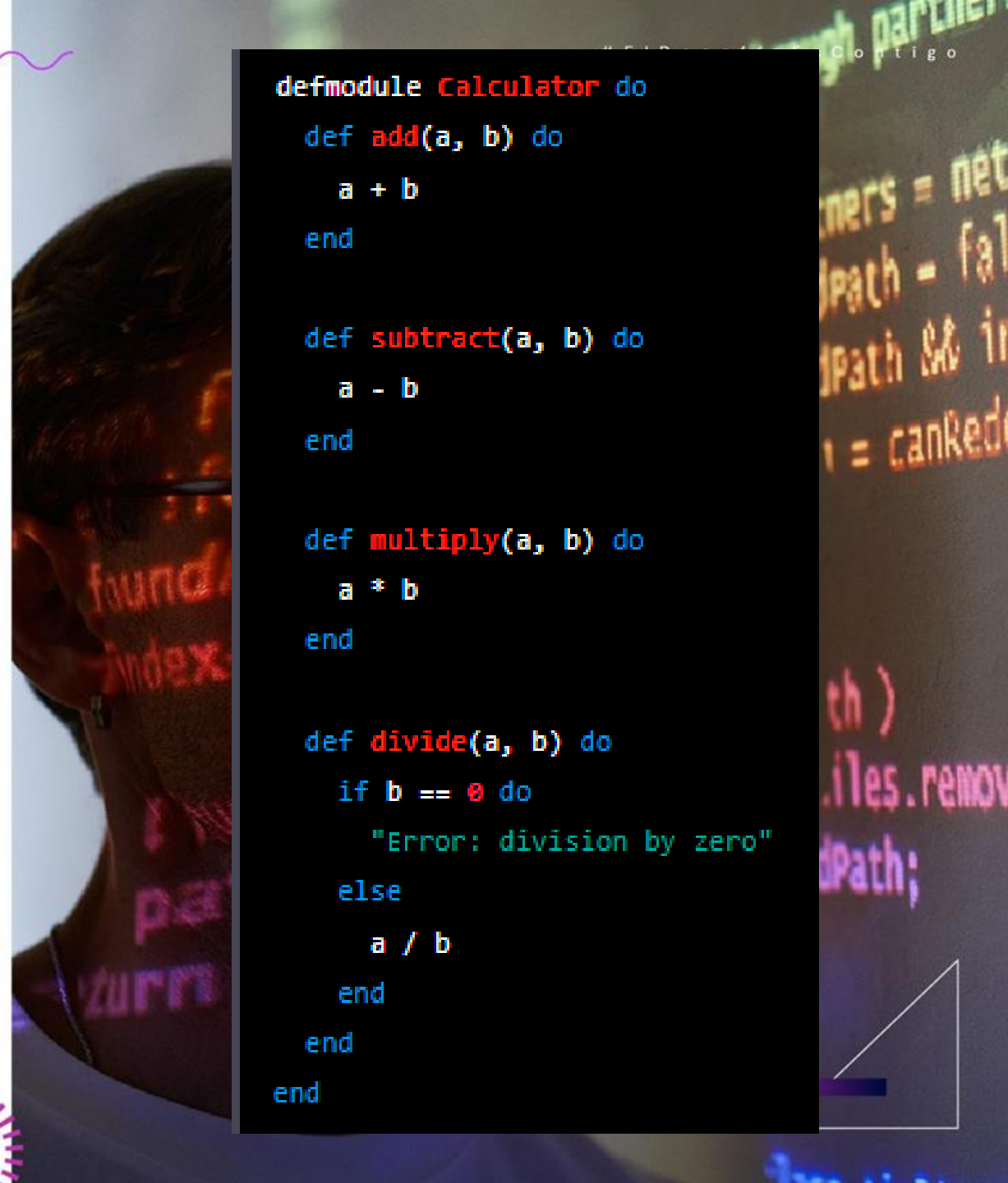
Programación funcional

Módulos, Alias e Import



Agrupando funcionalidad

- Los módulos definen un espacio de nombres para funciones, macros y tipos.
- Se definen utilizando la macro defmodule, seguida del nombre del módulo y de cualquier atributo opcional del módulo.

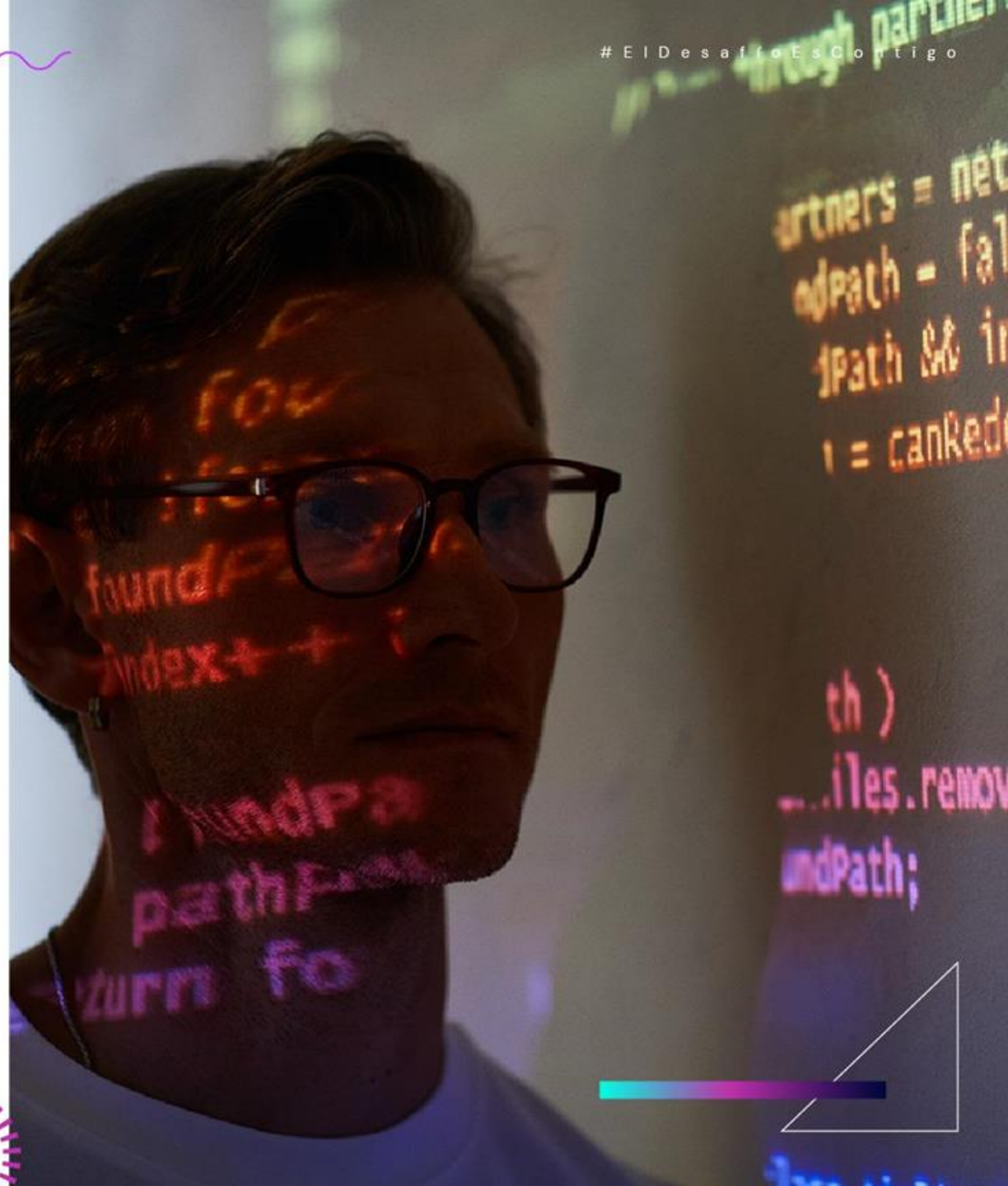


```elixir
defmodule Calculator do
  def add(a, b) do
    a + b
  end

  def subtract(a, b) do
    a - b
  end

  def multiply(a, b) do
    a * b
  end

  def divide(a, b) do
    if b == 0 do
      "Error: division by zero"
    else
      a / b
    end
  end
end
```

Alias

- Alias se utiliza para crear un nombre más corto para un módulo.
- Permite hacer referencia a un módulo utilizando un alias más corto en lugar del nombre completo del módulo.
- Los alias pueden crearse utilizando la palabra clave alias seguida del nombre completo del módulo y del nombre del alias.

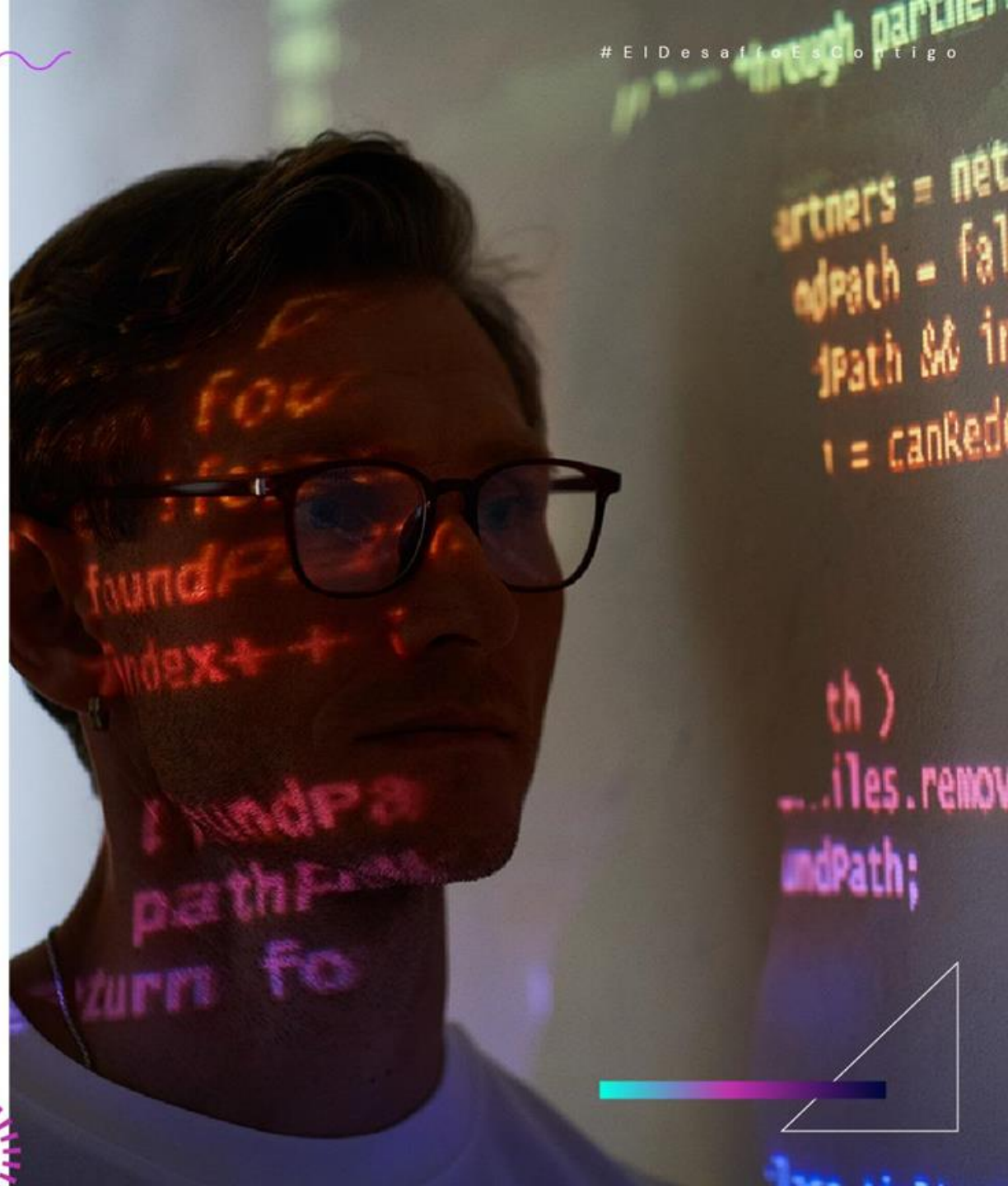
```
alias Enum, as: E
```



Import

- Permite traer funciones o macros de un módulo al espacio de nombres actual, haciéndolas disponibles sin necesidad de anteponerles el nombre del módulo.
- Puede utilizarse para importar todas las funciones de un módulo o funciones específicas.

```
import Enum, only: [map: 2, filter: 1]
```



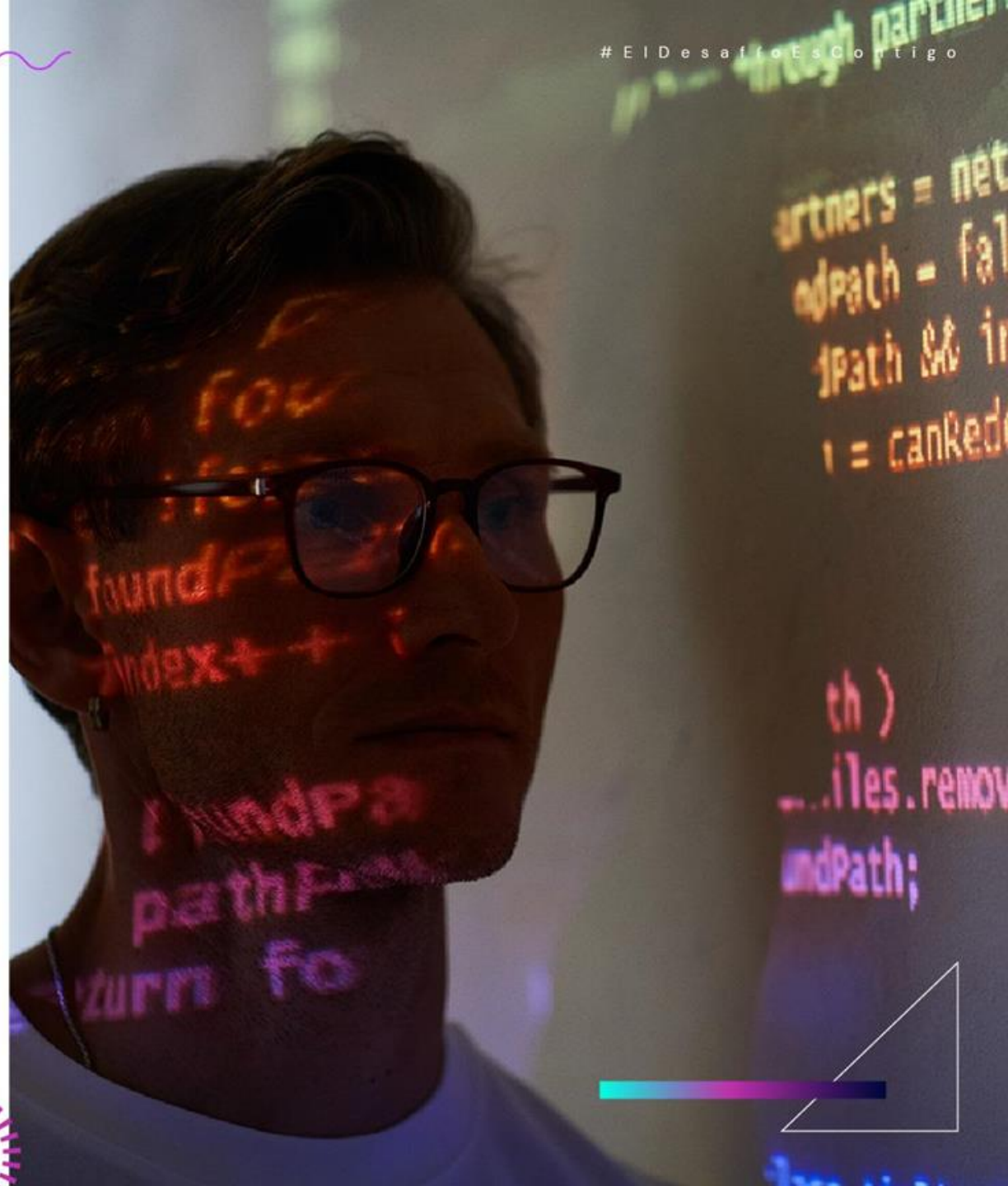
Ejemplo

1. Definición del módulo Math.Calculator.
2. Uso de alias para crear un ‘nickname’ más corto.
3. Uso de import para traer todas las funciones de Calc para no usar el prefijo que es necesario al llamar a las funciones.

```elixir
defmodule Math.Calculator do
  def add(a, b), do: a + b
  def subtract(a, b), do: a - b
  def multiply(a, b), do: a * b
  def divide(a, b), do: a / b
end

# Using alias to simplify the module name
alias Math.Calculator, as: Calc

# Using import to avoid module prefix
import Calc

# Calling the functions without module prefix
sum = add(2, 3)
diff = subtract(10, 5)
product = multiply(4, 6)
quotient = divide(15, 3)

IO.puts("Sum: #{sum}")
IO.puts("Difference: #{diff}")
IO.puts("Product: #{product}")
IO.puts("Quotient: #{quotient}")
```

Programación funcional

Es un paradigma de la programación, donde el software se compone de funciones que utilizan expresiones con estilo declarativo permitiendo devolver valores sin cambiar el estado del mismo.

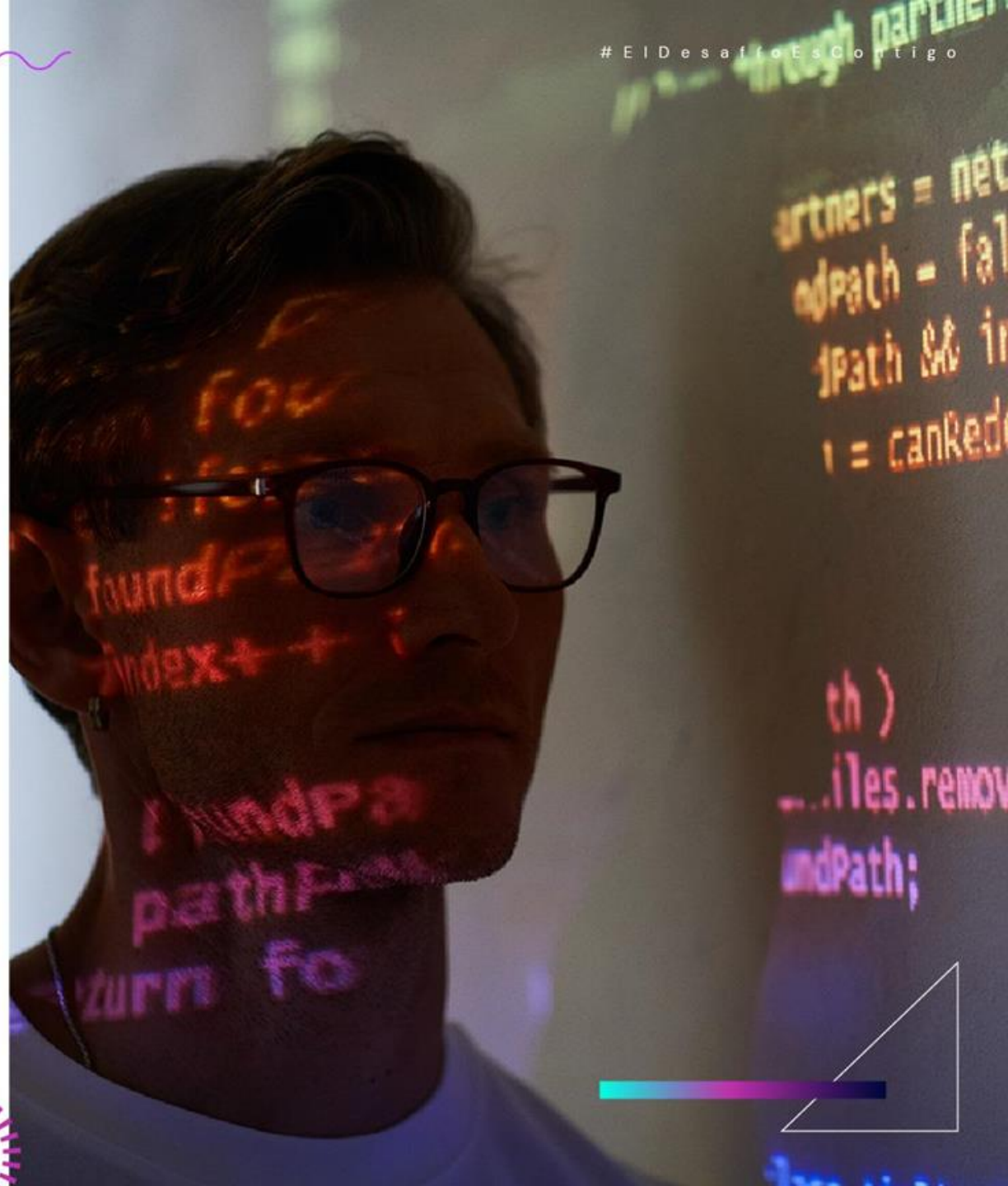


Funciones

- Pueden aceptar (o no) parámetros y devuelve un resultado al finalizar su ejecución.

- Las funciones con nombre se especifican por su nombre y su aridad.

```
24  @doc """
25  Multiply the two arguments together.
26  Return the product.
27  """
28  def multiplier(a, b) do
29    a*b
30  end
31
32  @doc """
33  Multiply the three arguments together.
34  Return the product.
35  """
36  def multiplier(a, b, c) do
37    a*b*c
38  end
```

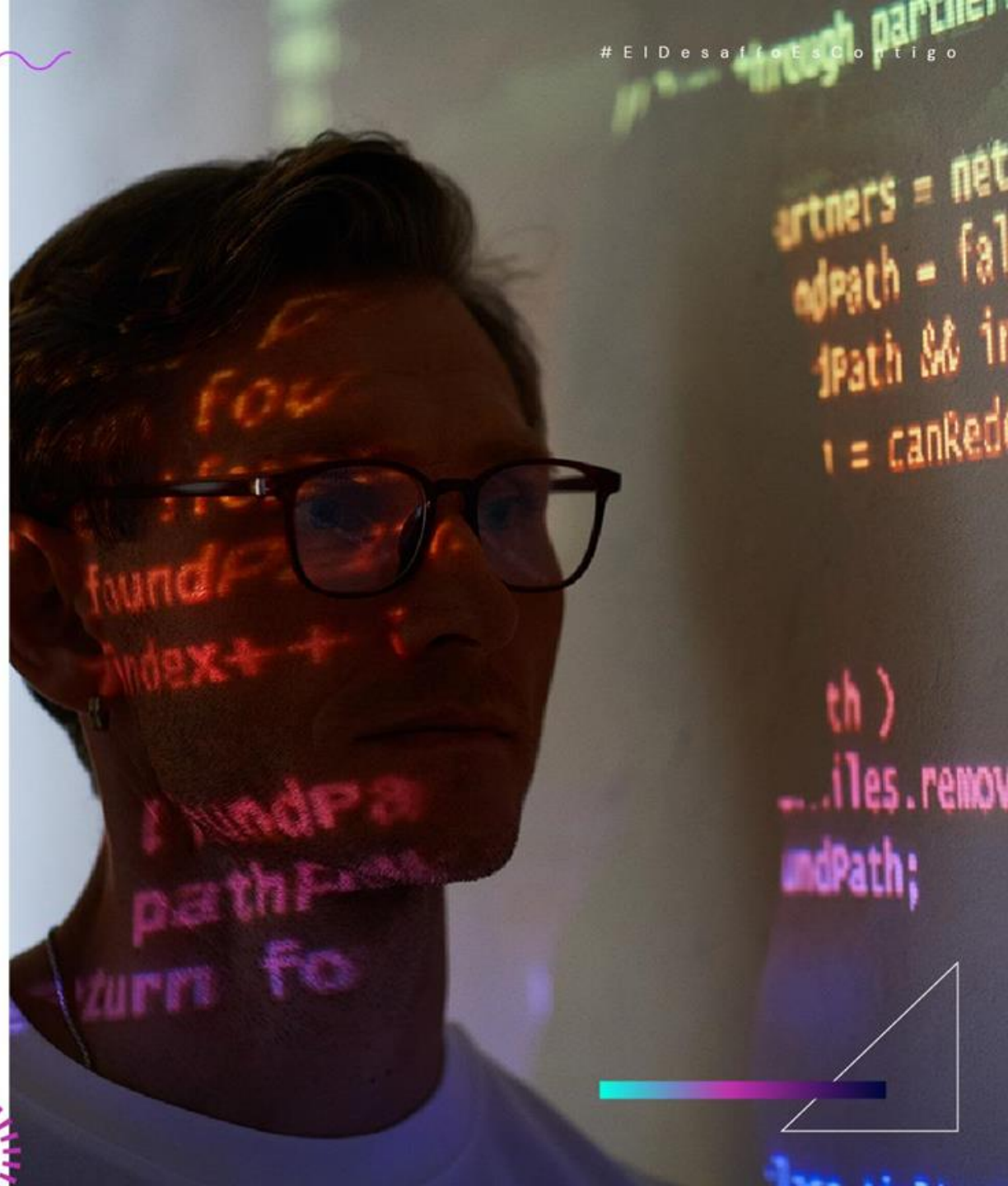


Funciones anónimas

- Pueden aceptar (o no) parámetros y devuelve un resultado al finalizar su ejecución.
- El carácter '&' es utilizado para reducir un poco la lectura al crear funciones anónimas.

```
iex> sum = fn (a, b) -> a + b end
iex> sum.(2, 3)
5
```

```
iex> sum = &(&1 + &2)
iex> sum.(2, 3)
5
```

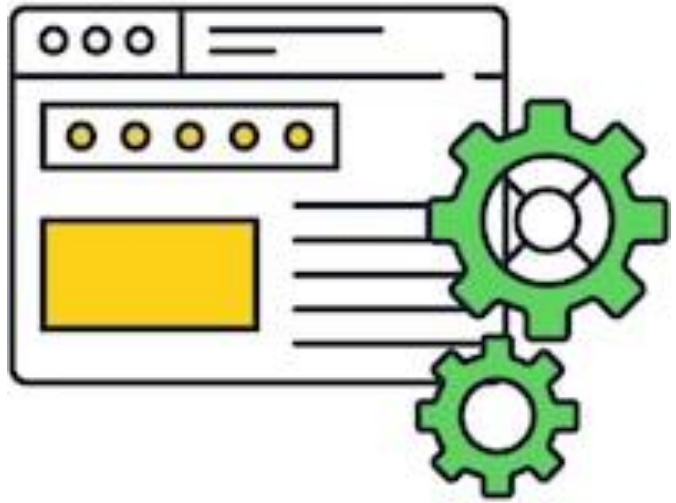


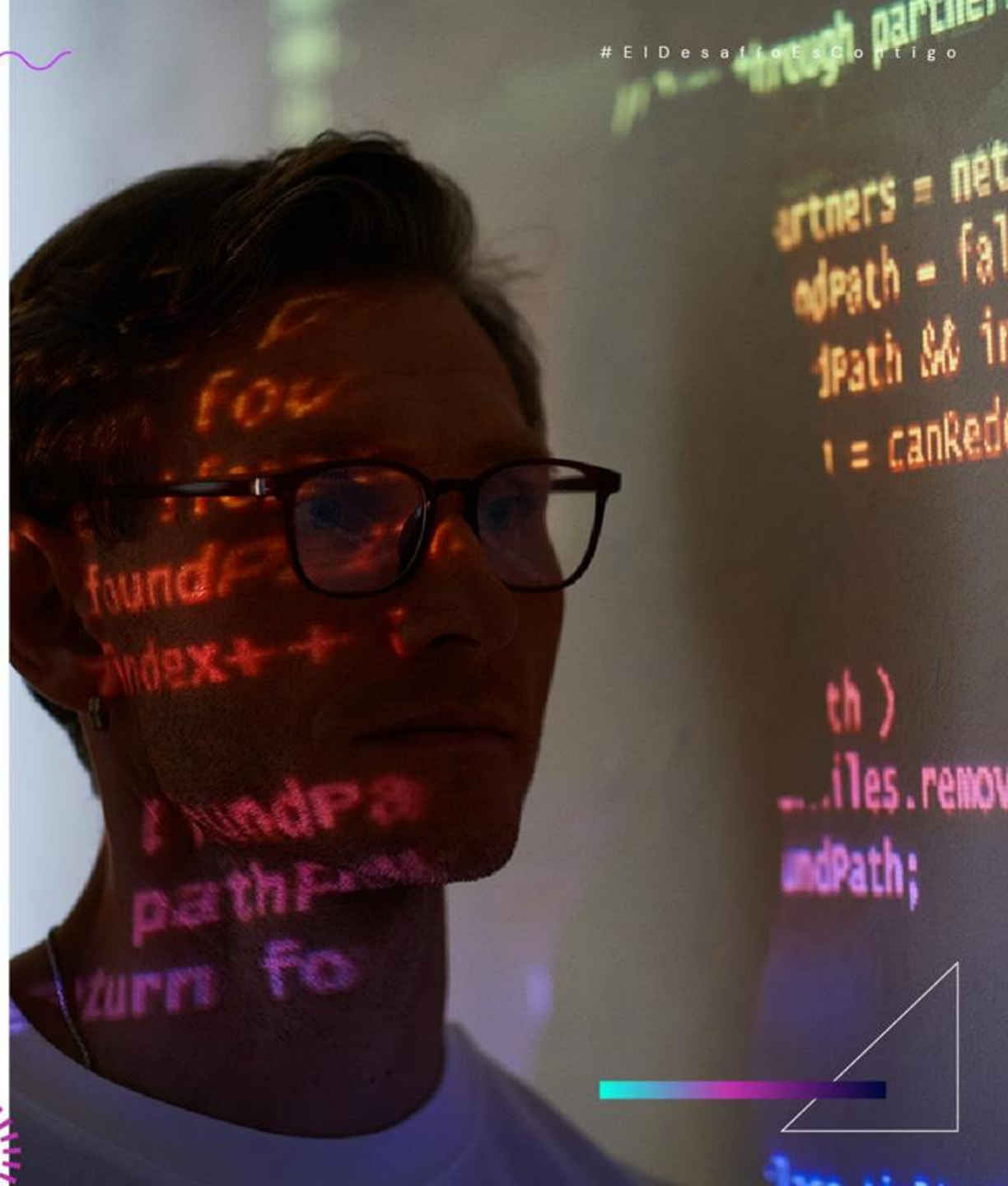
Programación funcional

Principios

¿Qué caracteriza puntualmente a la PF?

1. Inmutabilidad
2. Funciones puras
3. Funciones de primera clase
4. Funciones de orden superior
5. Recursión





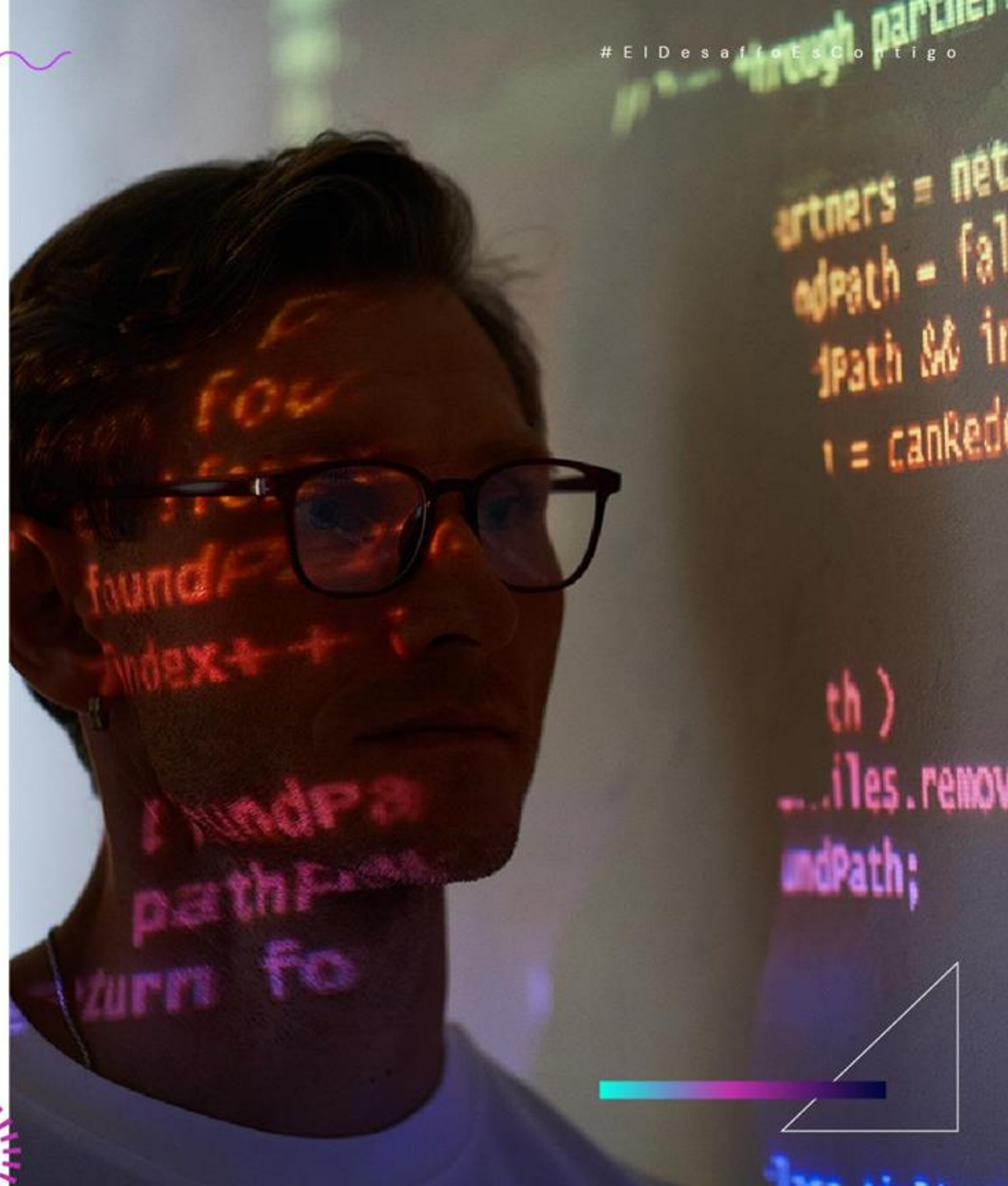
Inmutabilidad

Y entonces alguien dijo:

"Valor declarado no cambiará en ningún aspecto, ya sea en contenido y/o estado."

Pero alguien más se cuestionó:

"¿De qué sirve programar así si el mundo como tal no es del todo inmutable?



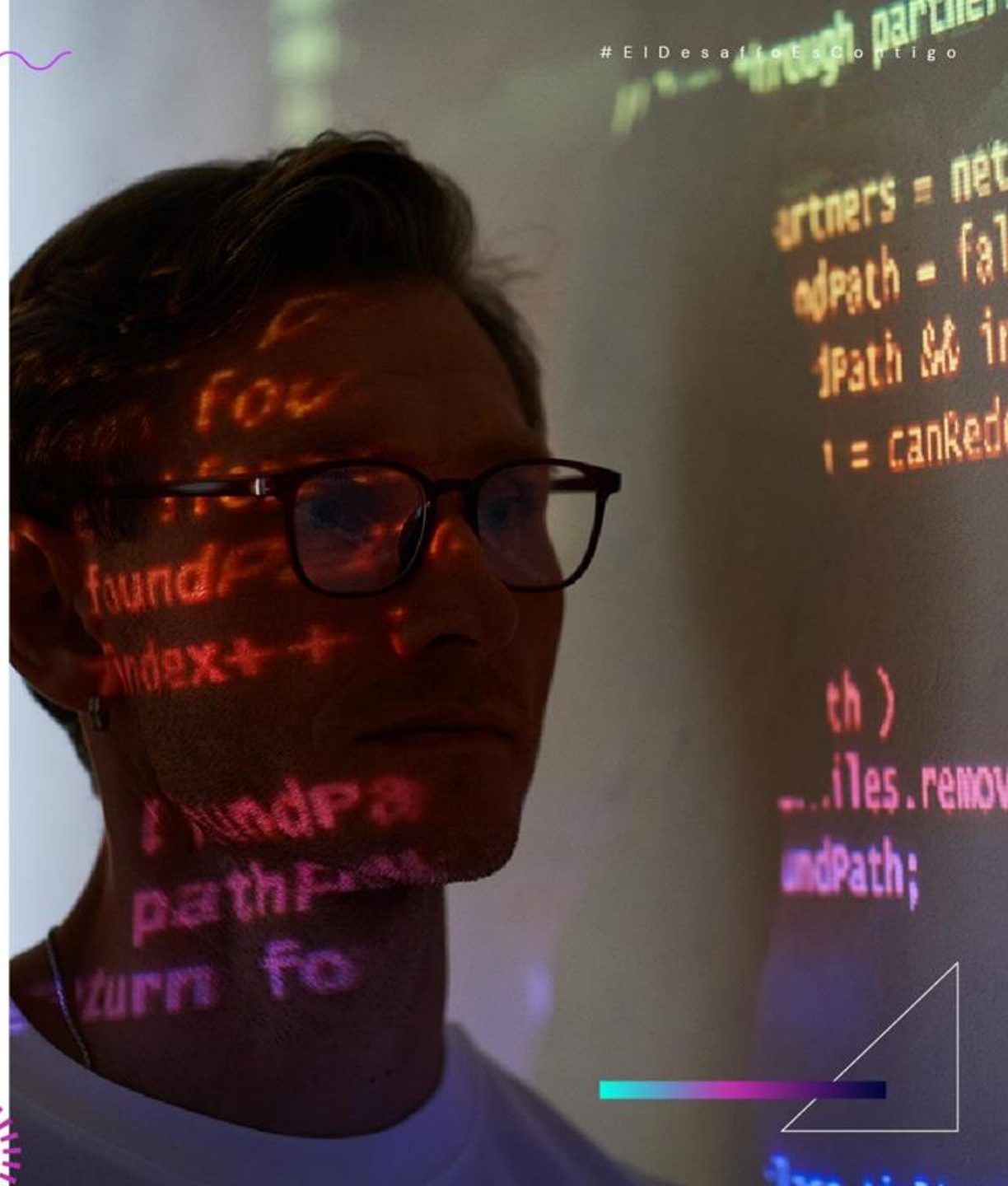
Inmutabilidad

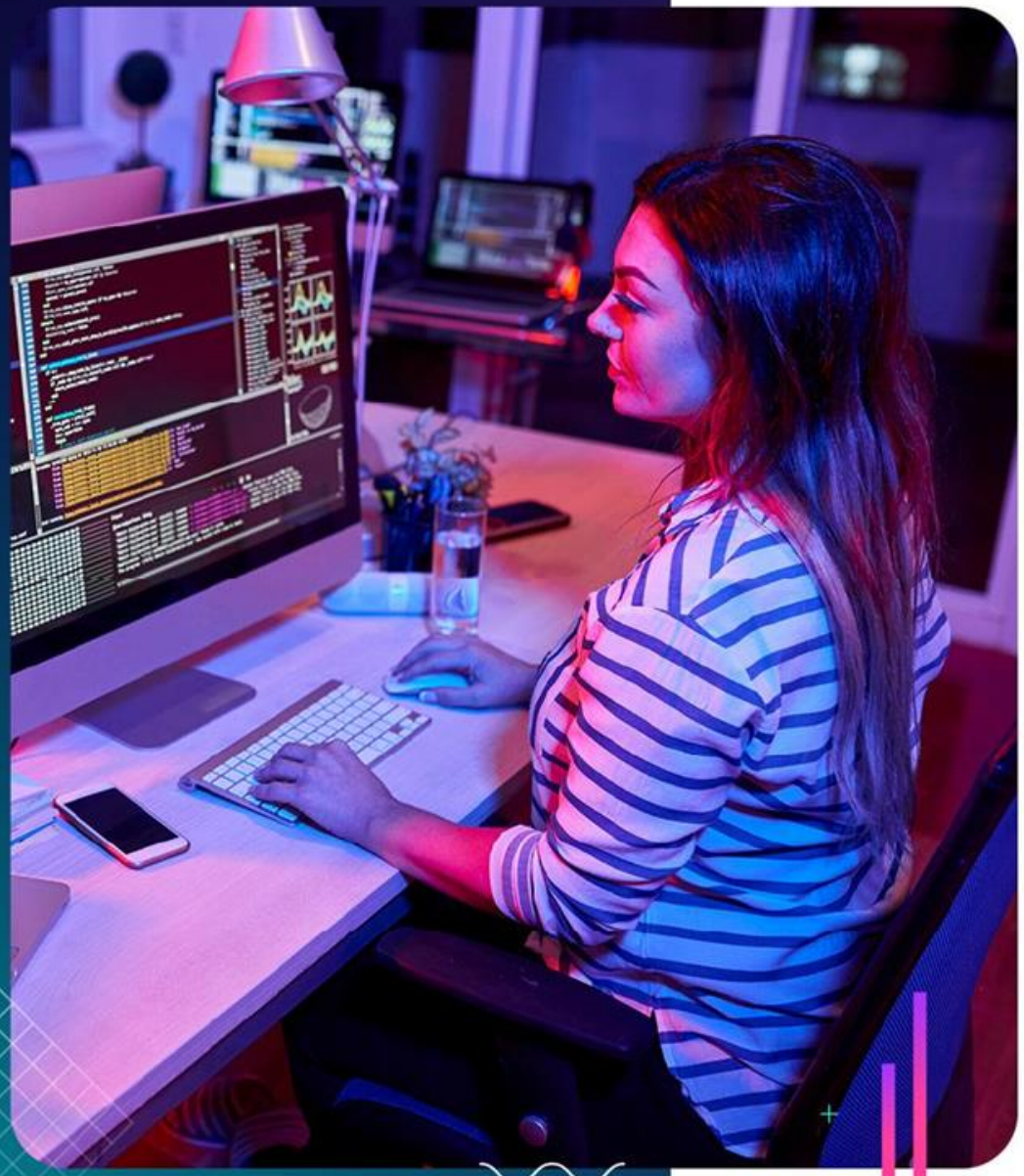
¿Asignar? No. ¿Enlazar? Sí.

```
1 | # Simple assignment
2 | a = 1
3 | b = 3
```

```
1 | # rebinding a variable
2 | a = 1
3 | a = 3
```

¿Confundido? Probablemente.





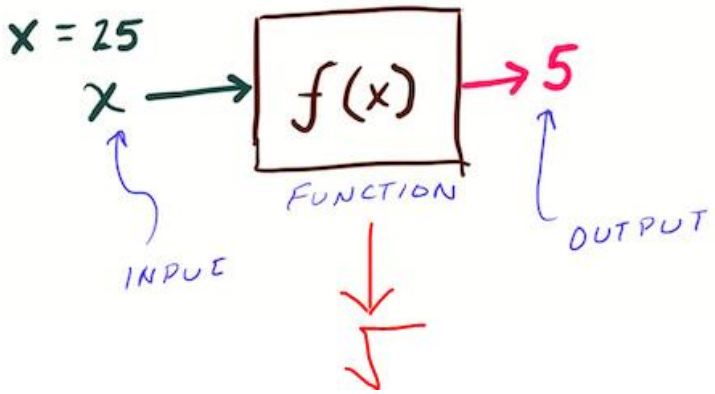
¿Qué nos ofrece la inmutabilidad?

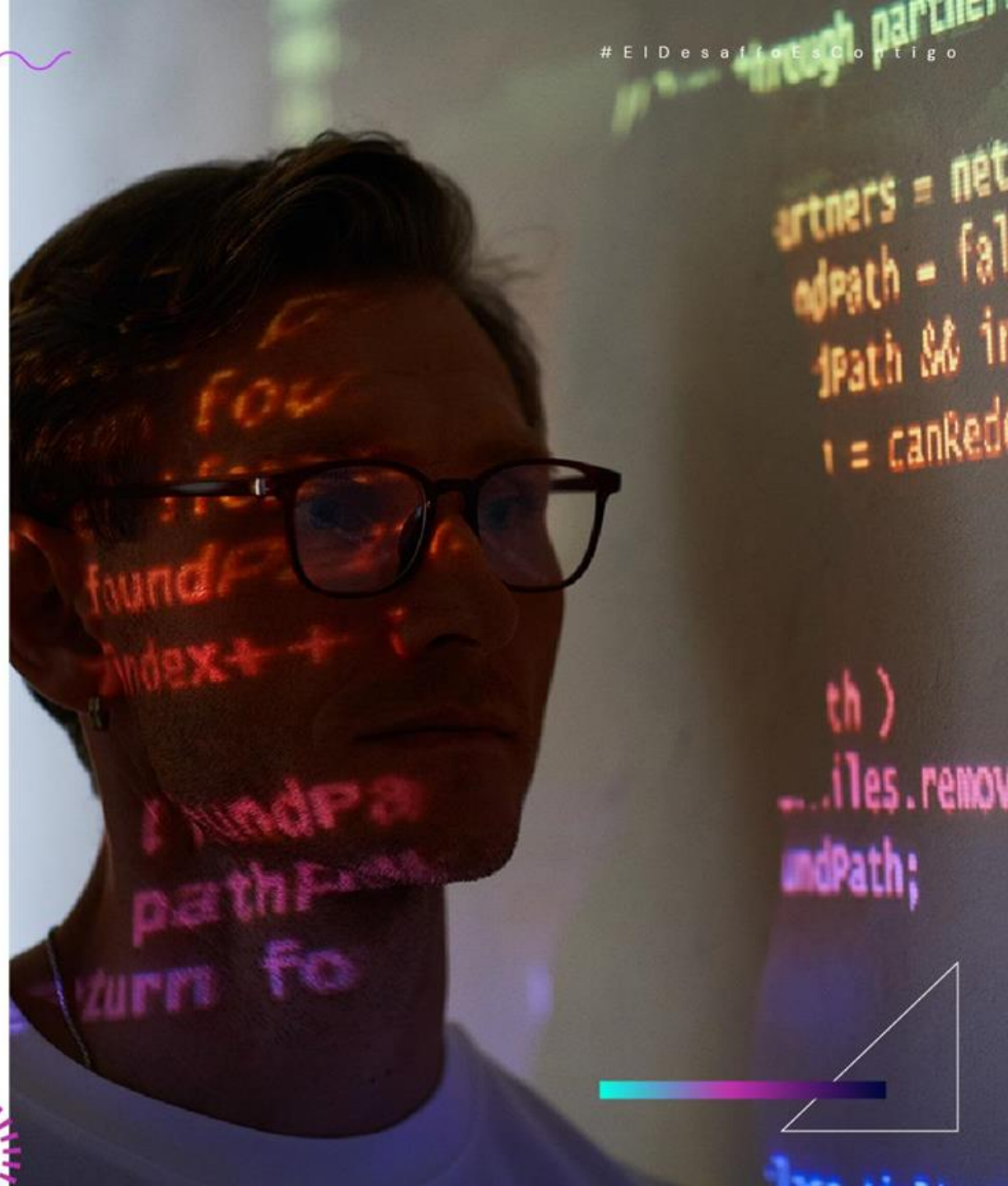
- Efectos secundarios inexistentes para apps multihilos
- Programación más simple y menos propensa a errores
- **No se sobre-escribe data, se transforma.**
- Mejor modularidad
- Poco mantenimiento

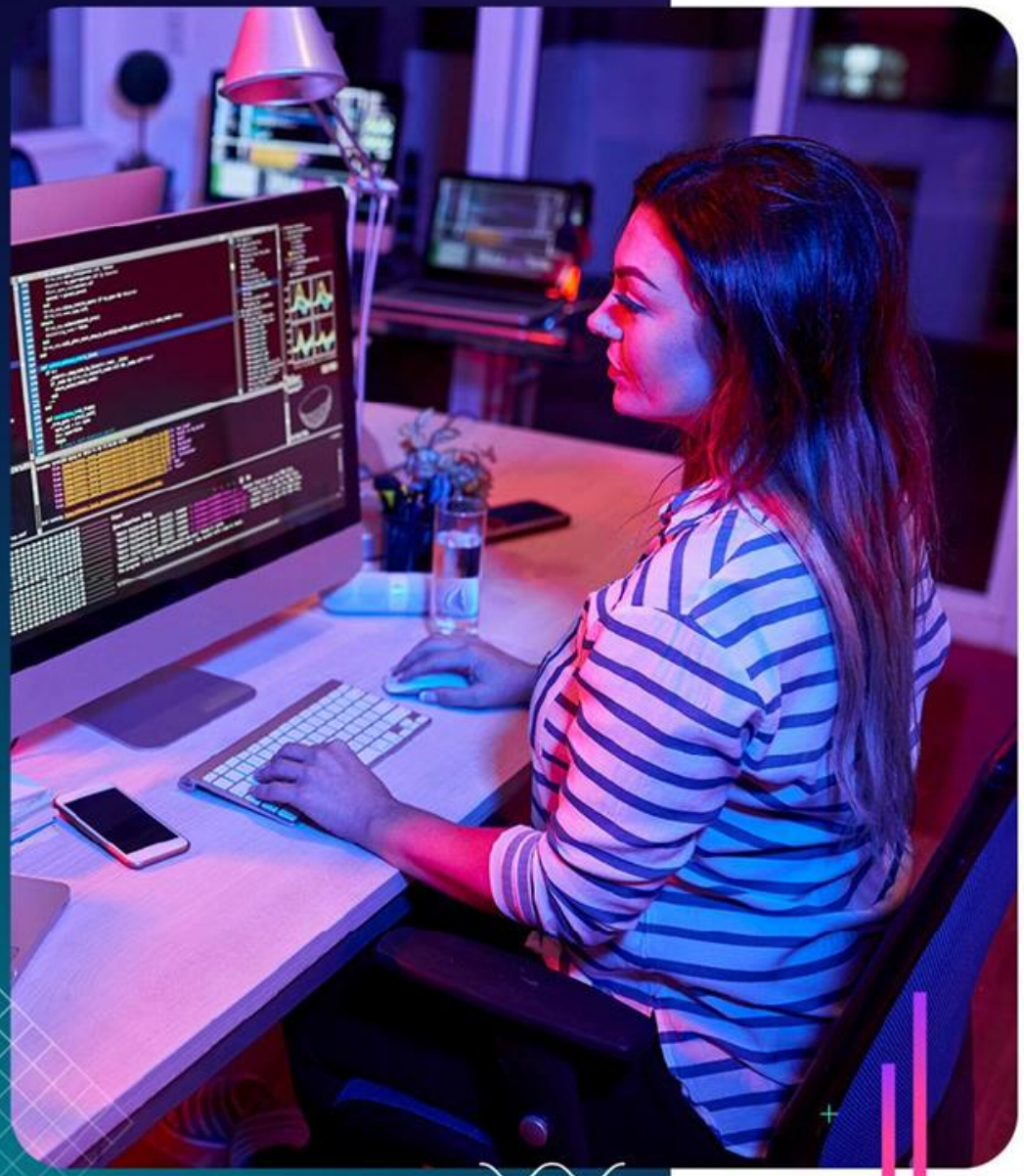
"La mutación es una realidad, el enfoque correcto es la mutación disciplinada"

Funciones puras

“Determinísticas y sin efectos secundarios”

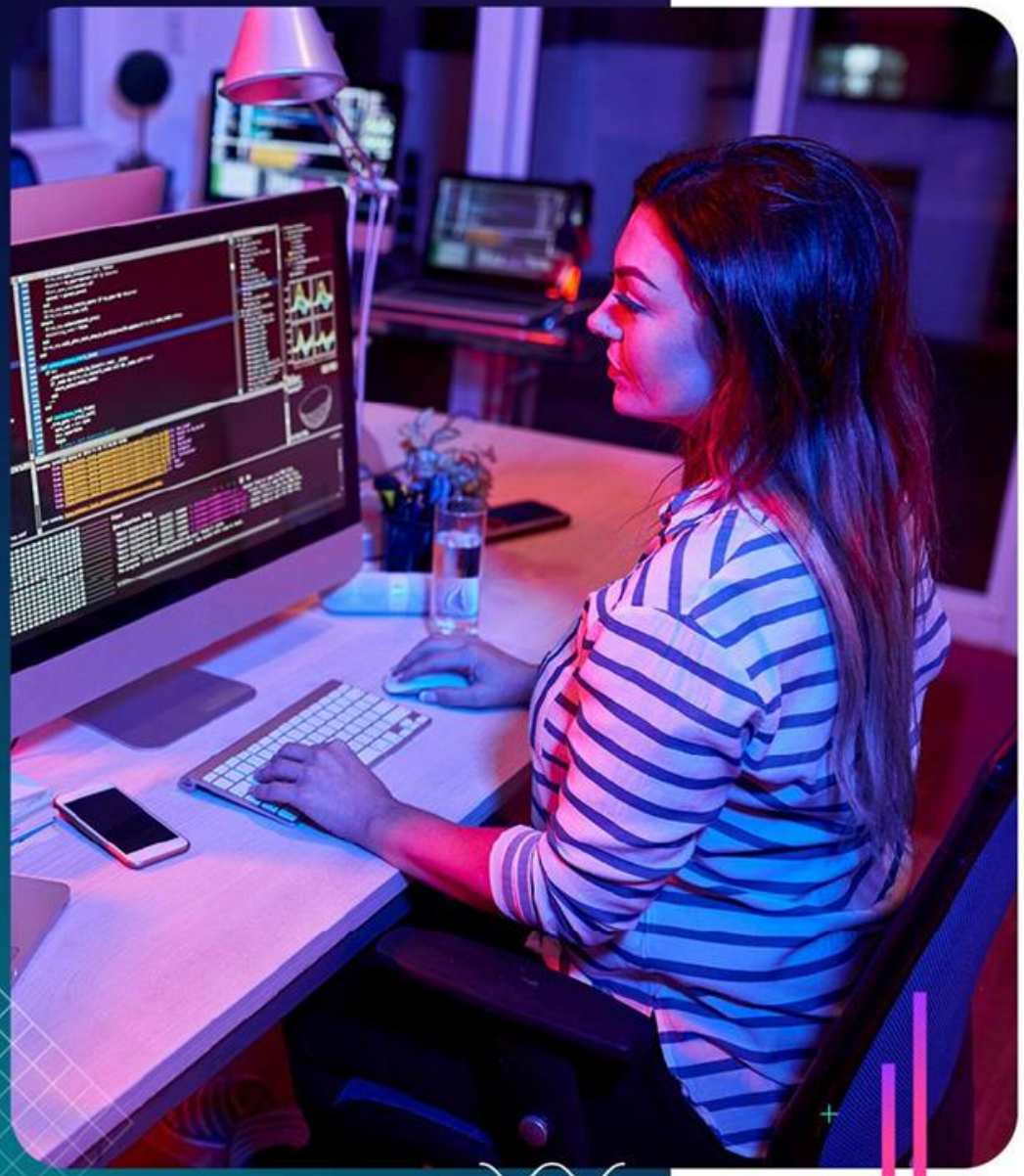






¿Y qué sucede con las funciones impuras?

- Las funciones se acoplan estrechamente con el entorno
- Aumenta la carga cognitiva del desarrollador
- Induce suposiciones de estado
- Aumenta la curva de aprendizaje de la base de código del desarrollador
- Condiciones de carrera
- Super-enemigo de la concurrencia
- Alta imprevisibilidad

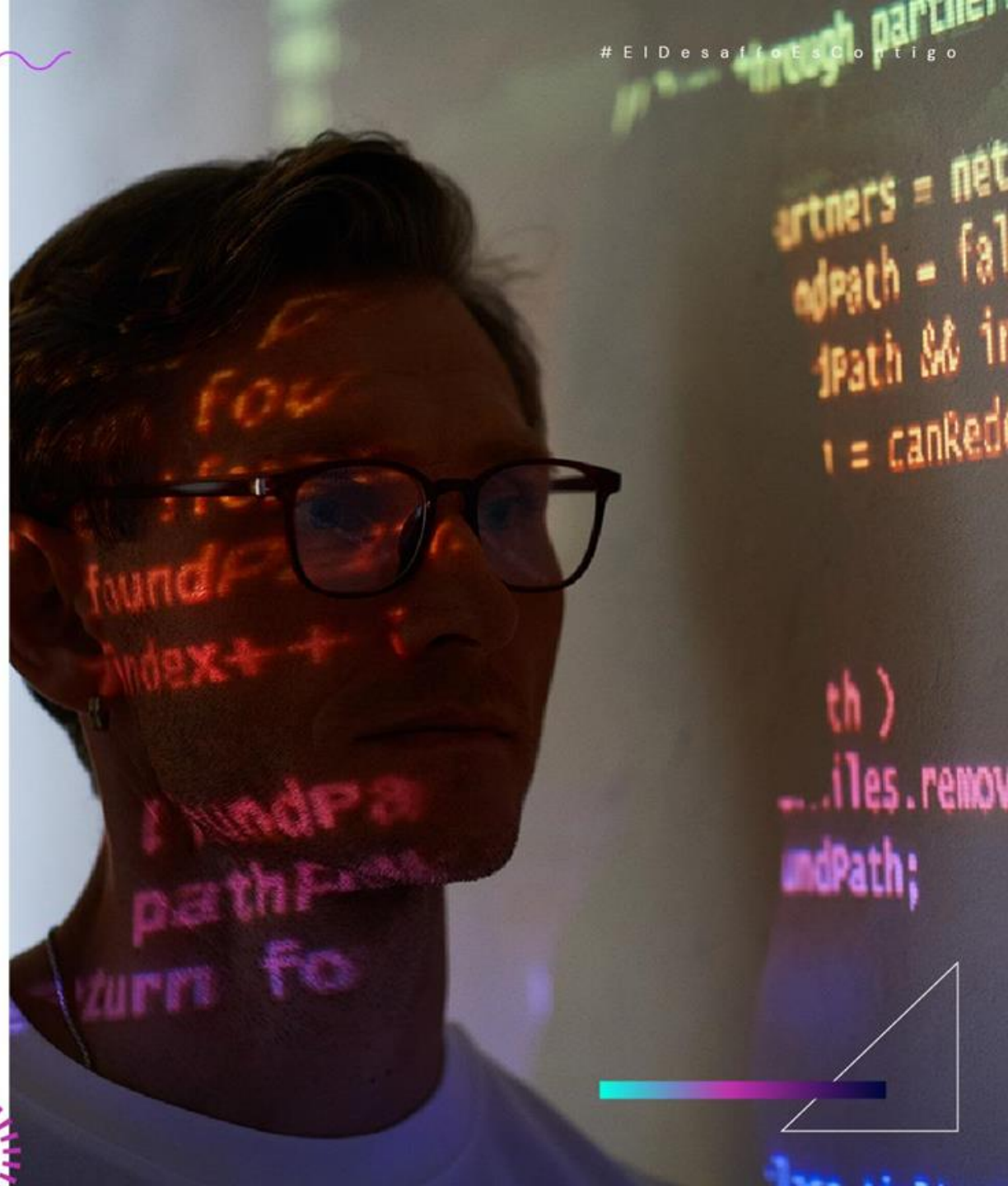


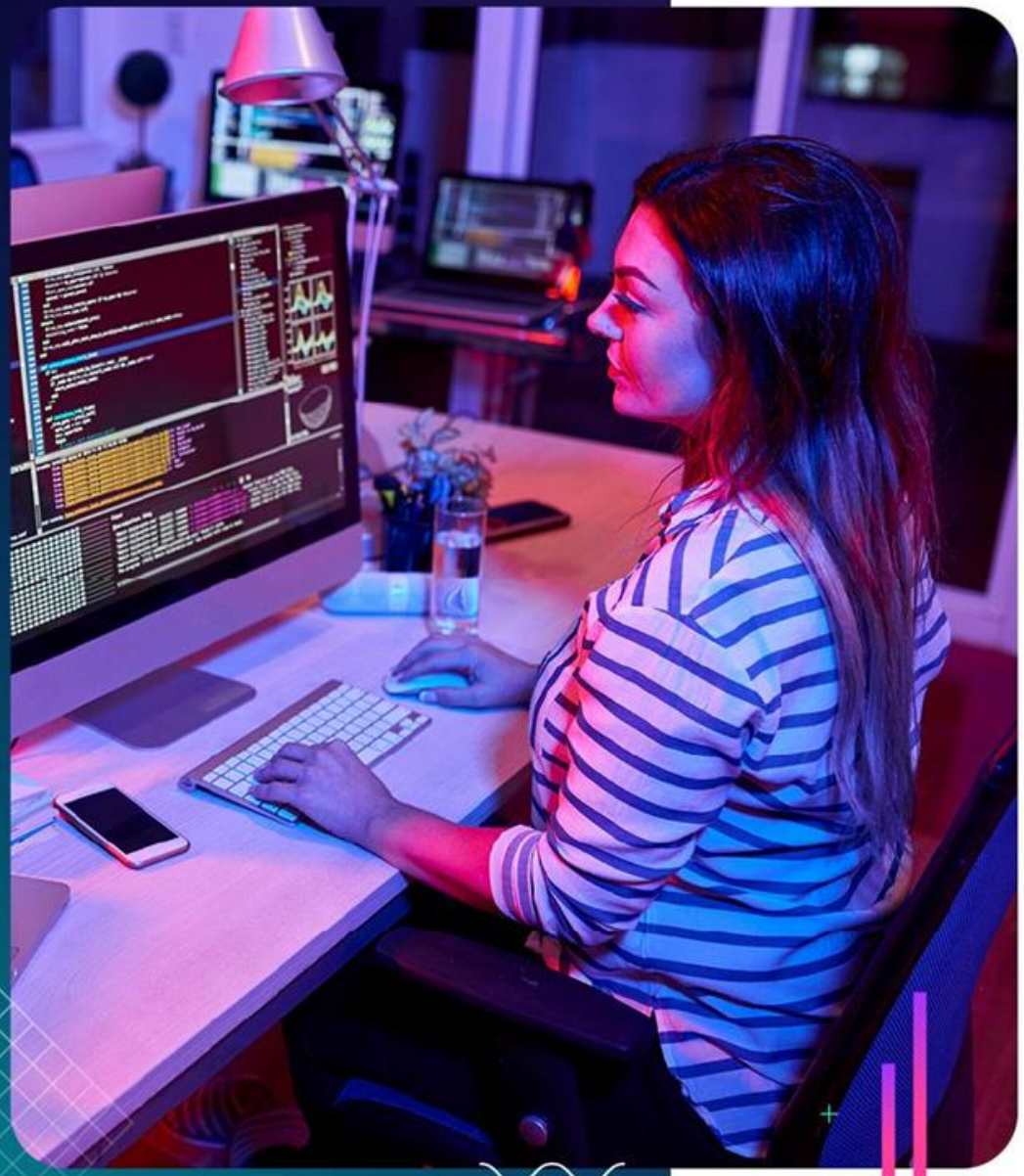
¿Qué nos ofrece las funciones puras?

- Independencia
- Facilidad de leer, testear y de mantener
- Inmutabilidad
- Predecibles
- Aplicables para ejecución en paralelo

Funciones de primera clase

> ***"Ni el cielo es el límite al crear o utilizar funciones"***

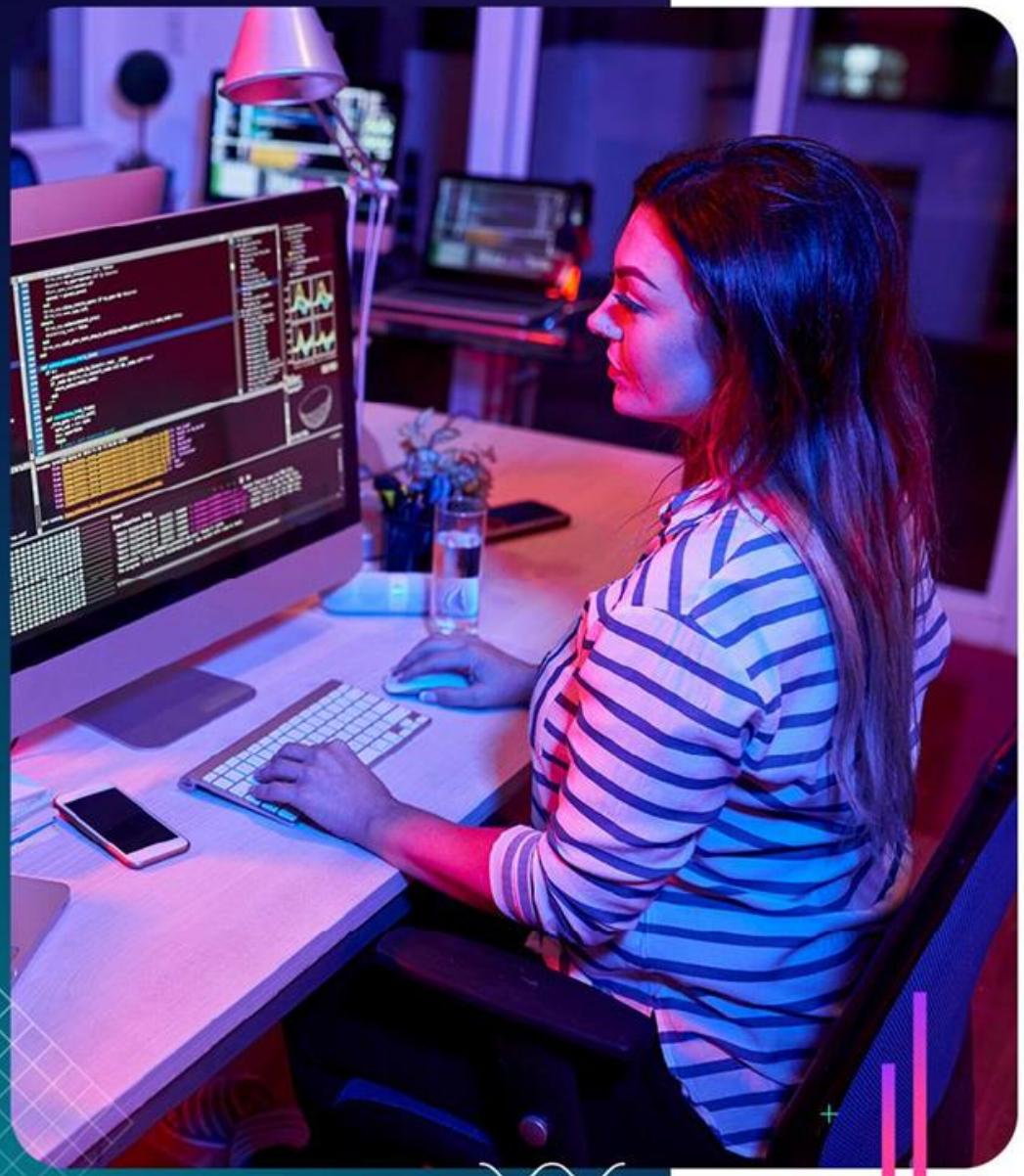




¿Qué ofrecen las funciones de primera clase?

- Asignación a variables regulares

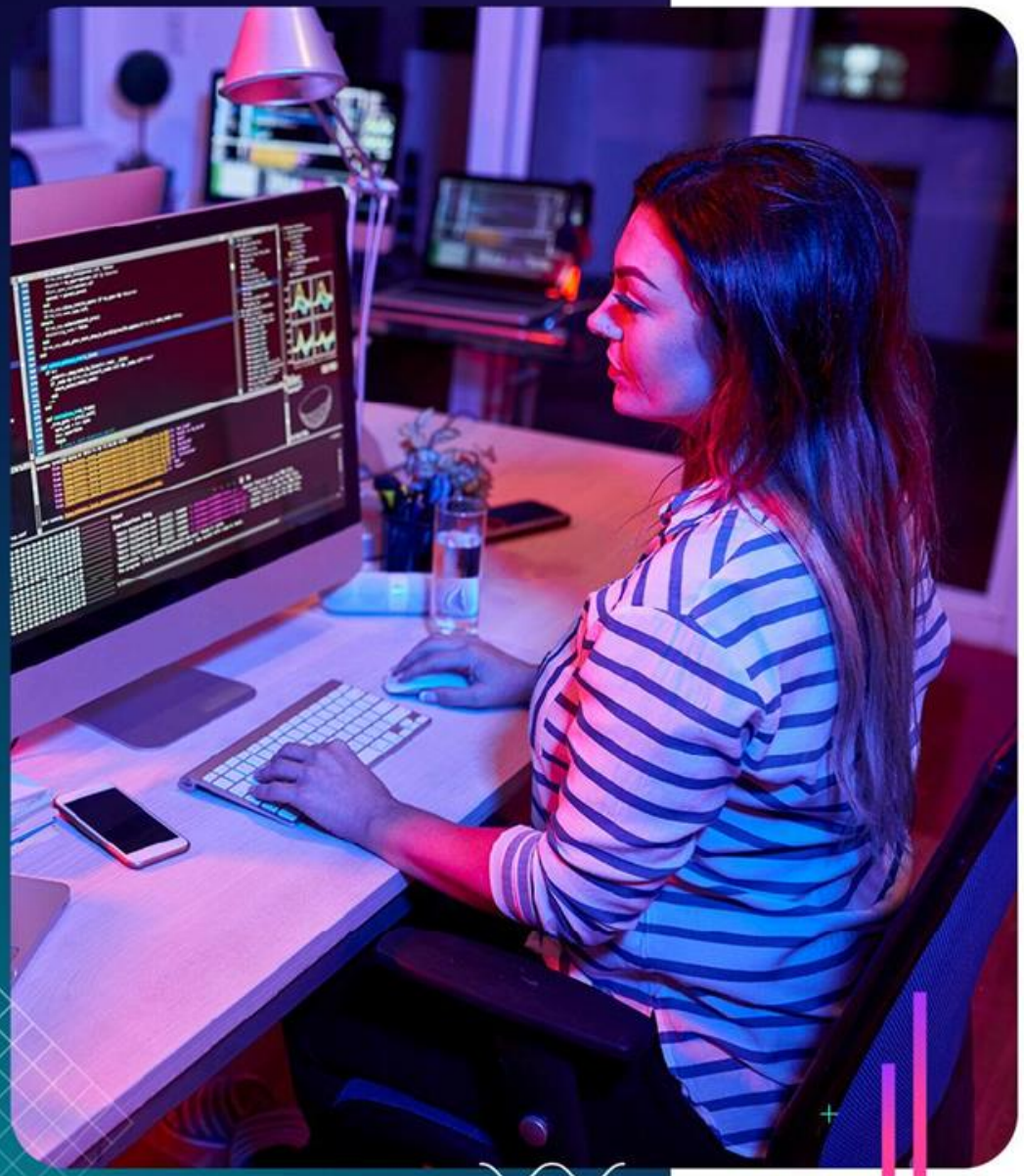
```
const greet   = (name) => `Hello ${name}`
// ... other primitive data types

greet('John') // Hello John
```



¿Qué ofrecen las funciones de primera clase?

- Se pasan como argumentos a las funciones

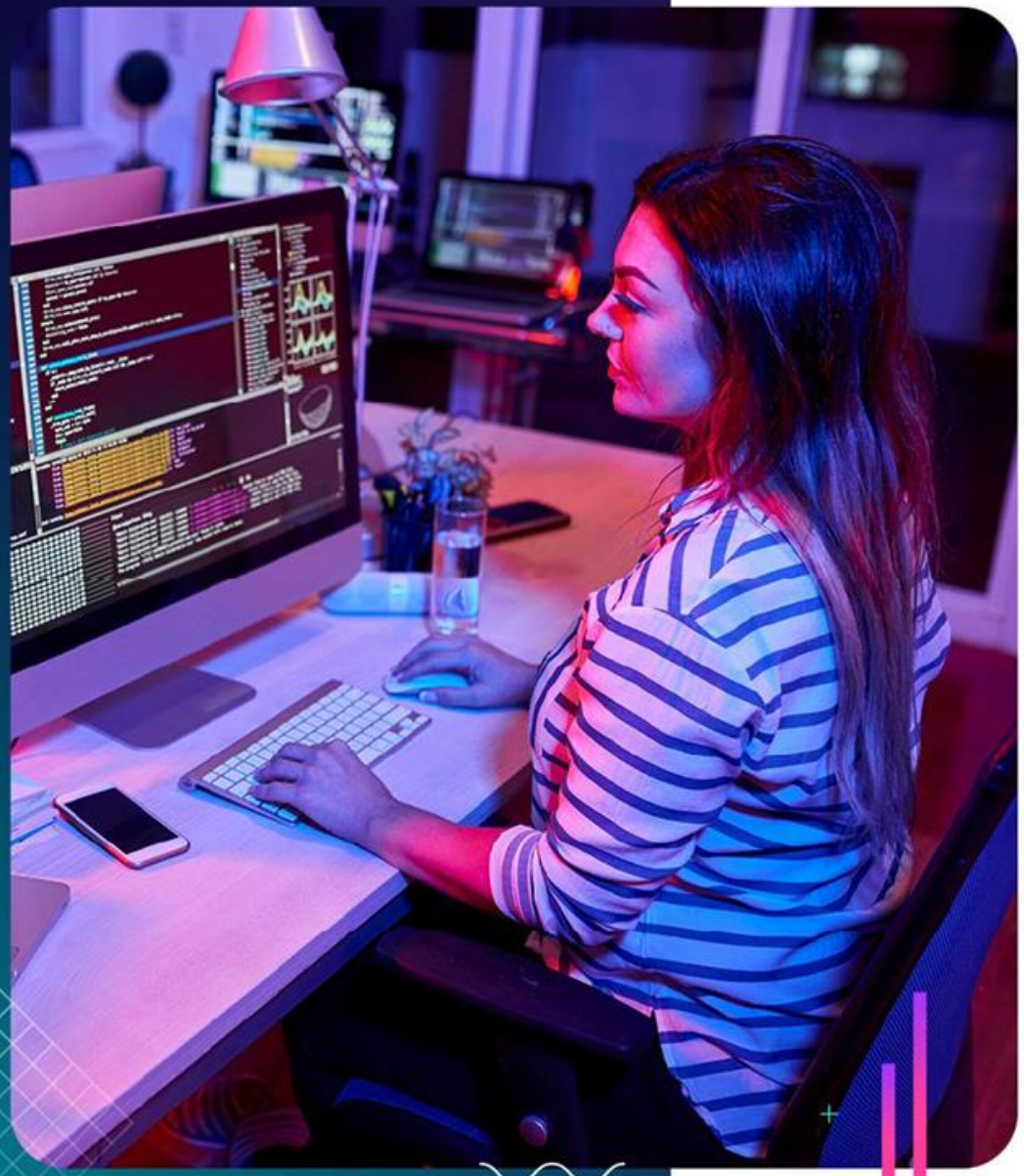
```
const nums = [1, 2, 3, 4, 5]

const addOne = (n) => n + 1

const addedOne = nums.map(addOne) // [2, 3, 4, 5, 6]
```



¿Qué ofrecen las funciones de primera clase?

- Se devuelven como resultados de funciones

```
const makeCounter = () => {
    let count = 0
    return () => ++count
}

const counter = makeCounter()

counter() // 1
counter() // 2
counter() // 3
counter() // 4
```



¿Qué ofrecen las funciones de primera clase?

- Se incluyen en cualquier estructura de datos

```
const wakeUp = name => `${name}, wake up early!`
const takeShower = name => `${name}, take shower!`
const workout = name => `${name}, workout!`
const shutUp = name => `${name}, shut up!`

const morningRoutines = [
    wakeUp,
    takeShower,
    workout,
    shutUp
]

morningRoutines.forEach(routine => routine('John'))
// John, wake up early!
// John, take shower!
// John, workout!
// John, shut up!
```

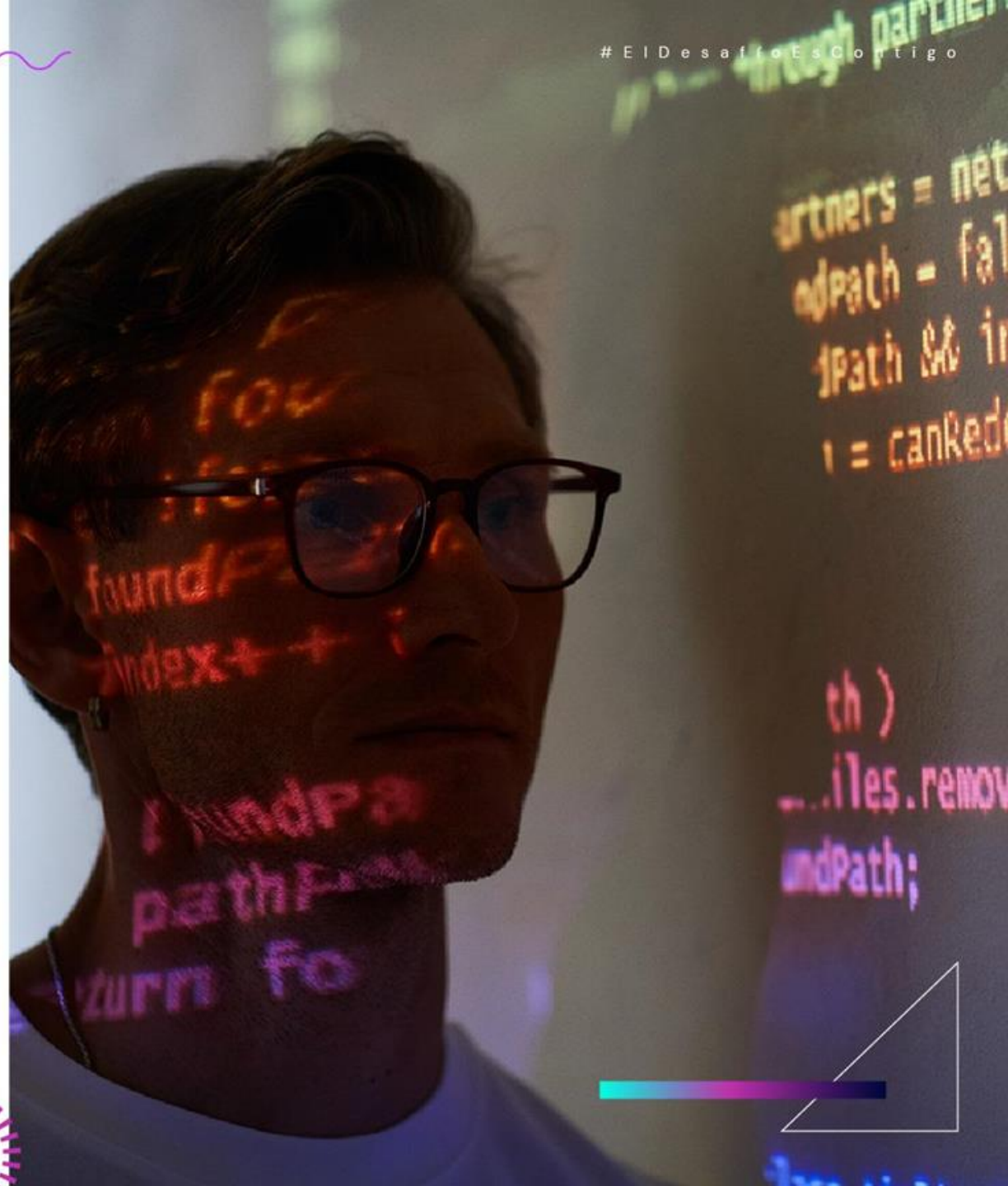
Funciones de orden superior

"Funciones que reciben otras funciones como argumentos"

```
const newArr = arr.map((val) => val ** 2);
console.log(newArr); // [ 1, 4, 9, 16, 25 ]
```

```
const dogs = animals.filter((animal) => animal.Species === "dog");

//Output:
[
    { name: "DexLuthor 🐕", Species: "dog" },
    { name: "Trenton 🐕", Species: "dog" },
    { name: "Joey 🐕", Species: "dog" },
];
```

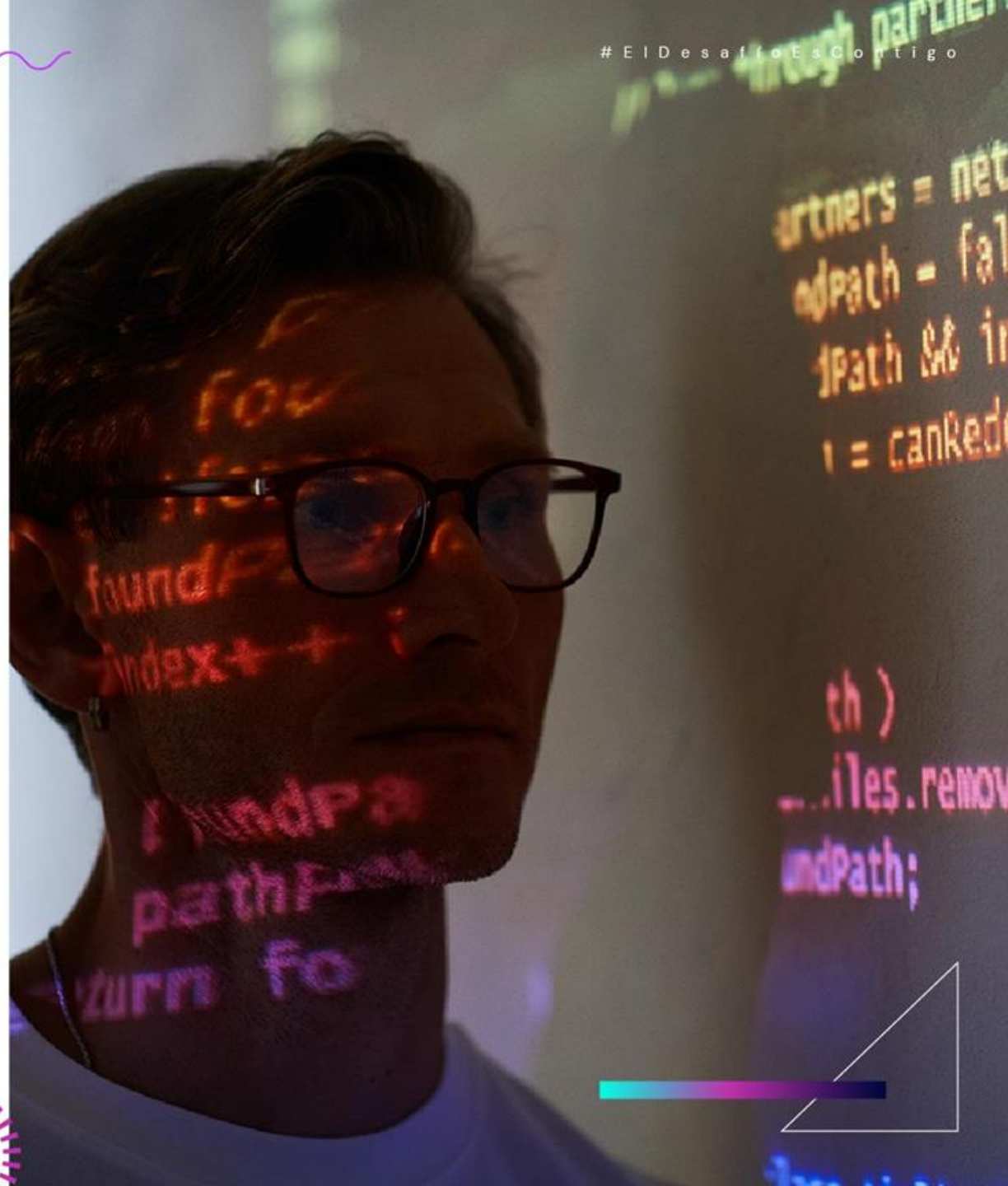
```
const avgSalary =
    salaries.reduce((avg, employee) => avg + employee.salary, 0) /
    salaries.length;


// Output
console.log(avgSalary); // 116250
```

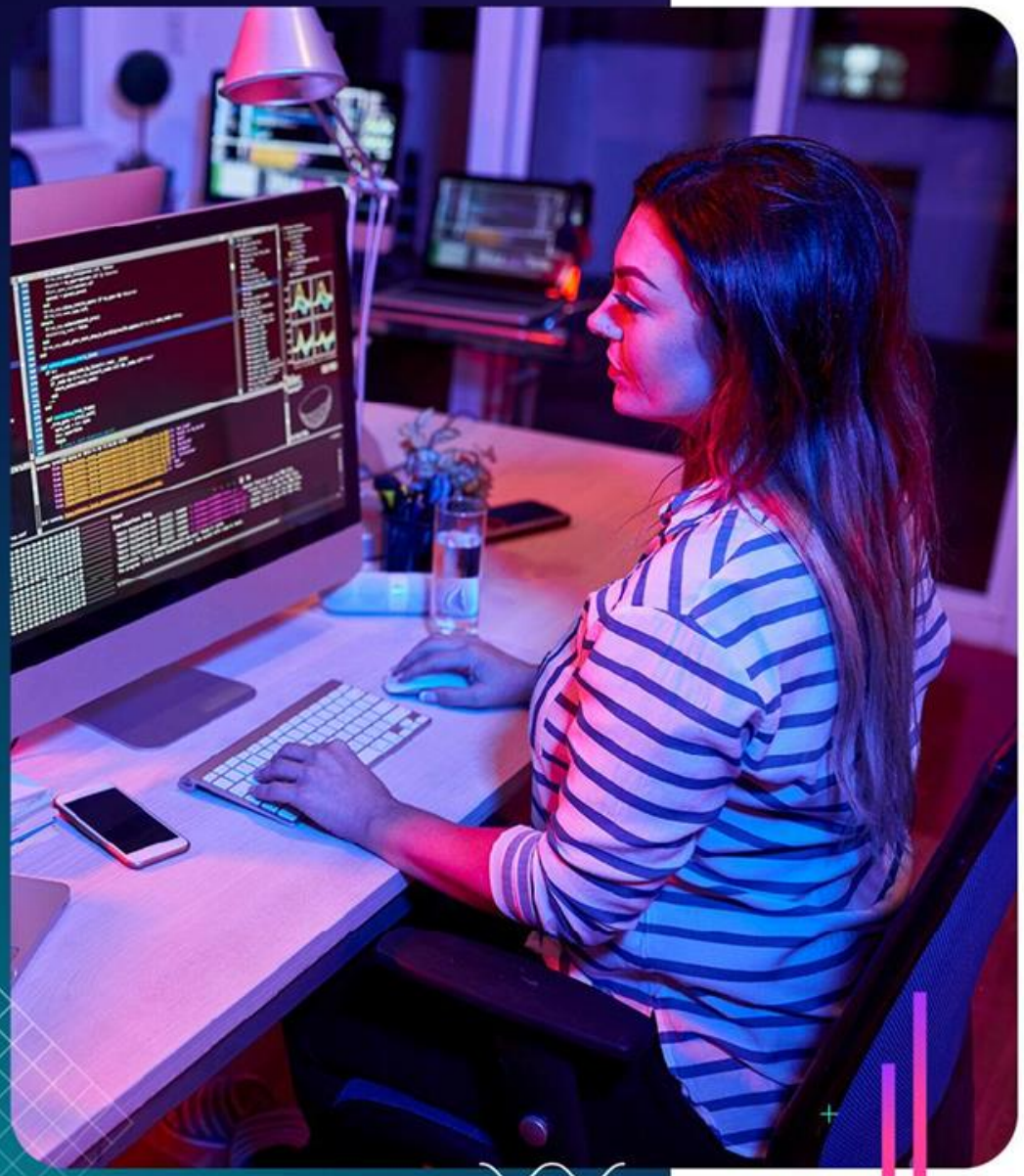


Recursión

```python
def fact(n):
    if n == 1 or n == 0:
        return 1

    else:
        return n * fact(n-1)

```

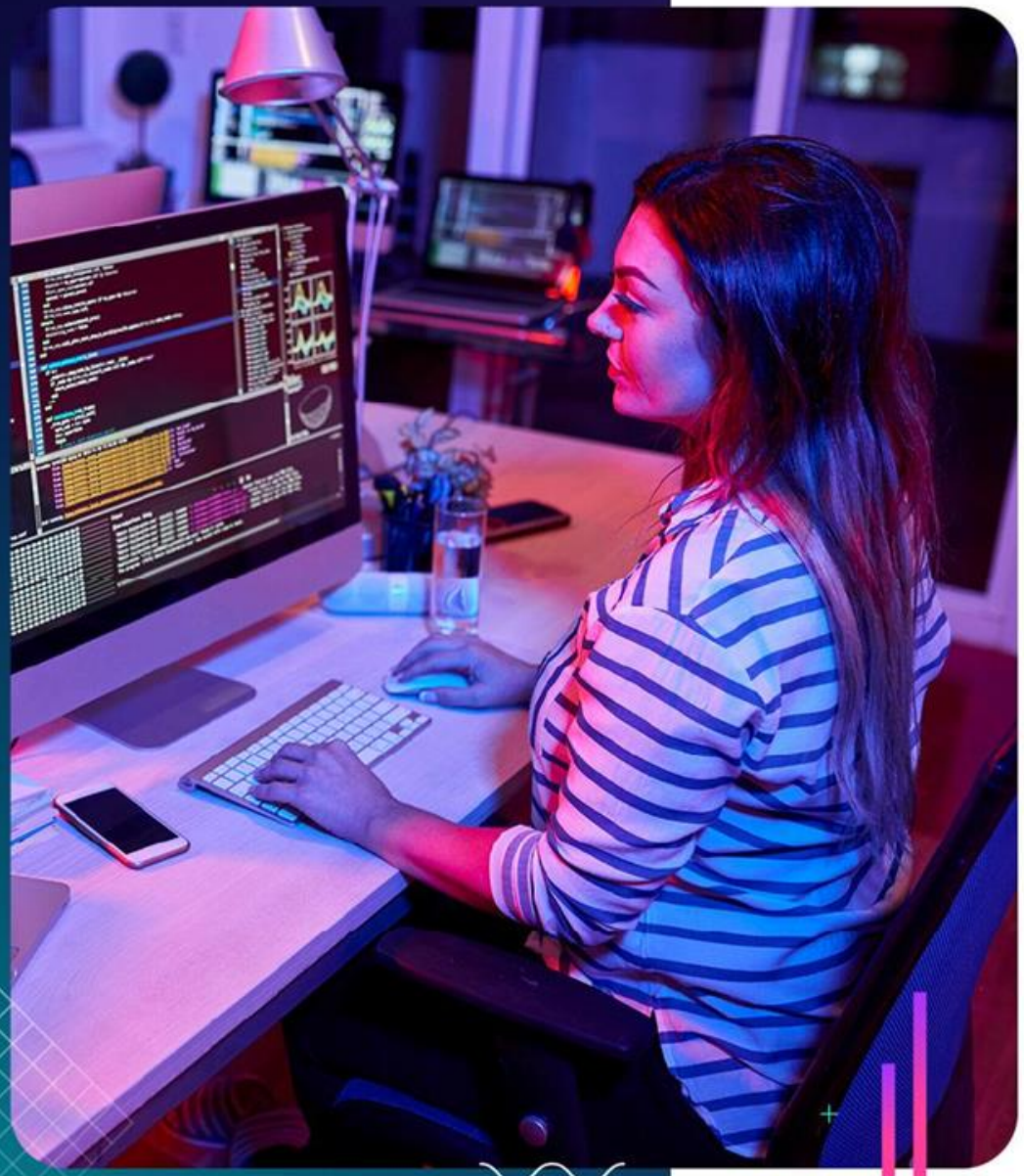
“Soy una función que se llama así mismo para calcular un resultado.”





¿Qué componen a una función recursiva?

- Caso base: La condición final de una función recursiva llamada con éxito.
- Caso terminal: Un condicional que se llama si algo va mal o el estado que determina la finalización del proceso de recursión. Esto evita un bucle infinito.



¿Cómo entiendo una función recursiva?

- Identifique siempre el caso base de la función antes que cualquier otra cosa.
- Pasar argumentos a la función que alcanzarán inmediatamente el caso base.
- Identifique los argumentos que ejecutarán al menos una vez la llamada a la función recursiva.

Recursión

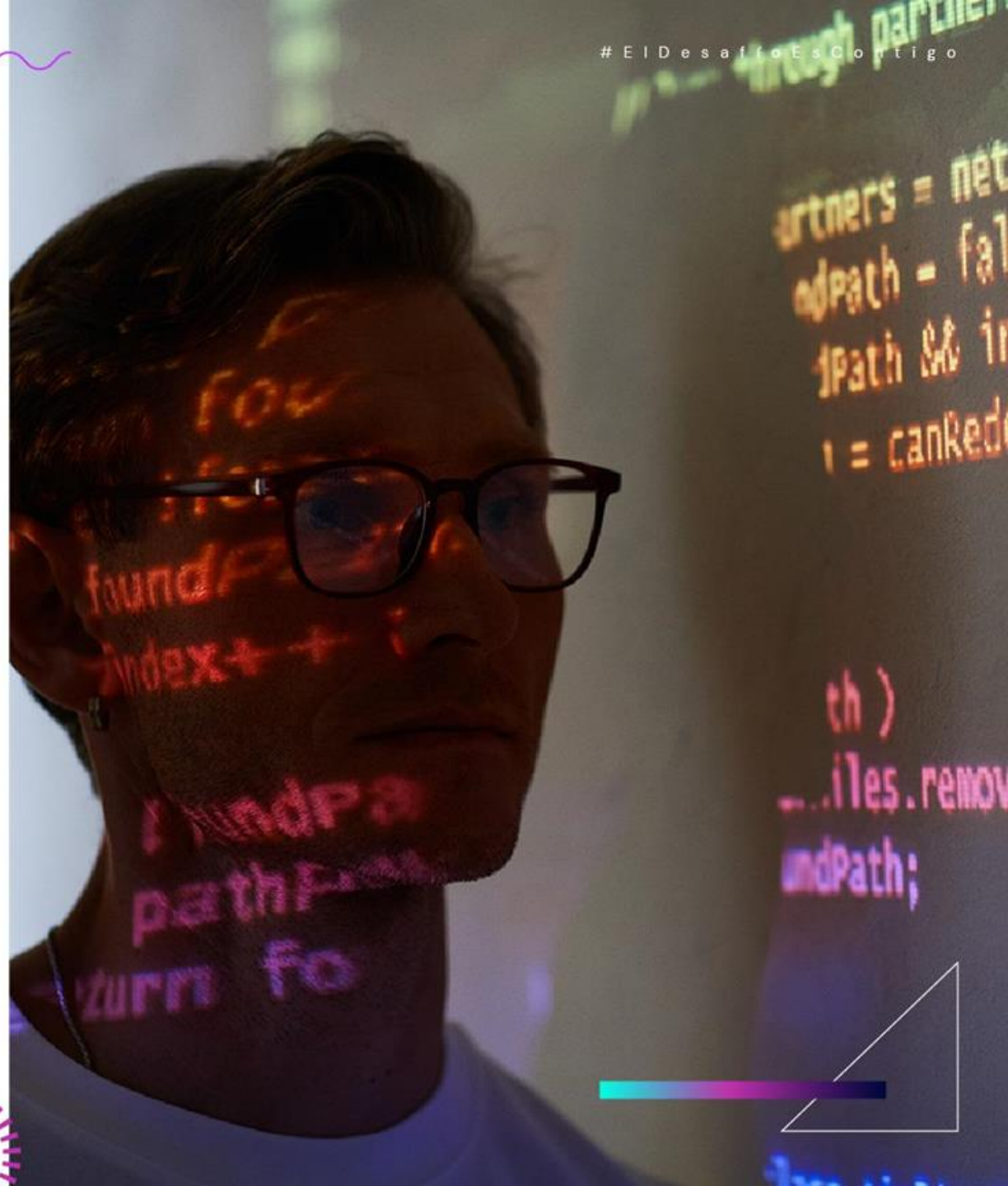
Función para calcular el factorial de un número n:

Partiendo del caso base:

```
1  def fact(n):
2      if n == 1 or n == 0:
3          return 1
4
```

Haciendo luego el llamado recursivo

```
1  def fact(n):
2      if n == 1 or n == 0:
3          return 1
4
5      else:
6          return n * fact(n-1)
7
```



Conclusiones

- Elixir es un lenguaje de programación funcional diseñado para facilitar la creación de sistemas distribuidos y tolerantes a fallos.
- La programación funcional es un paradigma de programación que hace hincapié en el uso de funciones puras y estructuras de datos inmutables para evitar efectos secundarios y hacer que el código sea más predecible y fácil de razonar.
- Las características de programación funcional de Elixir lo hacen idóneo para diversos casos de uso, como la programación web, el desarrollo backend, los sistemas distribuidos y la comunicación en tiempo real.

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

Preguntas & Respuestas

Elixir

Programación más funcional que nunca

Semana 2 – MC #4

SofkaU

#ElDesafíoEsContigo

Temas

01 Colecciones II

02 Expresiones

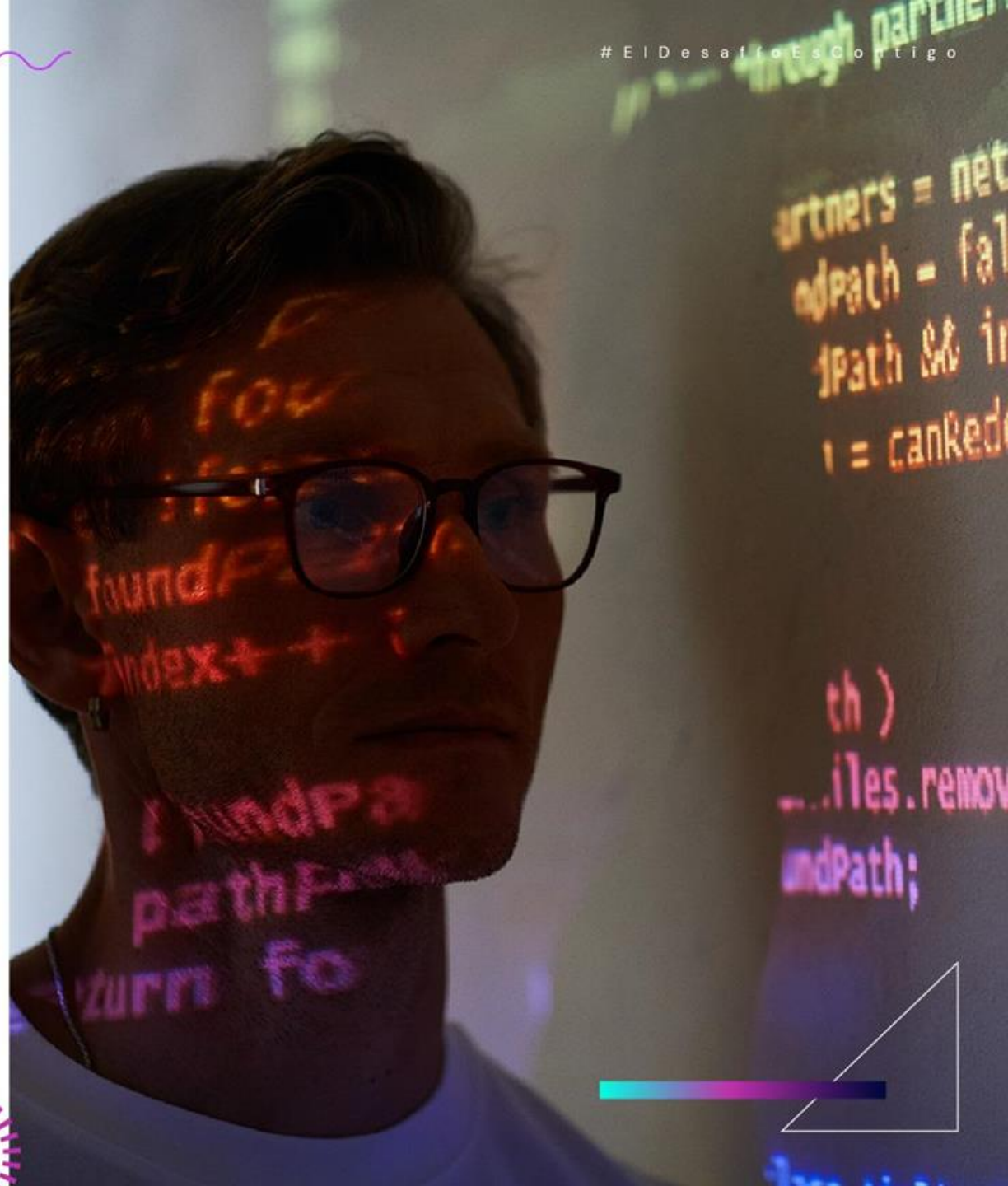
03 Preguntas y respuestas

Programación funcional

Expresiones

Expresiones

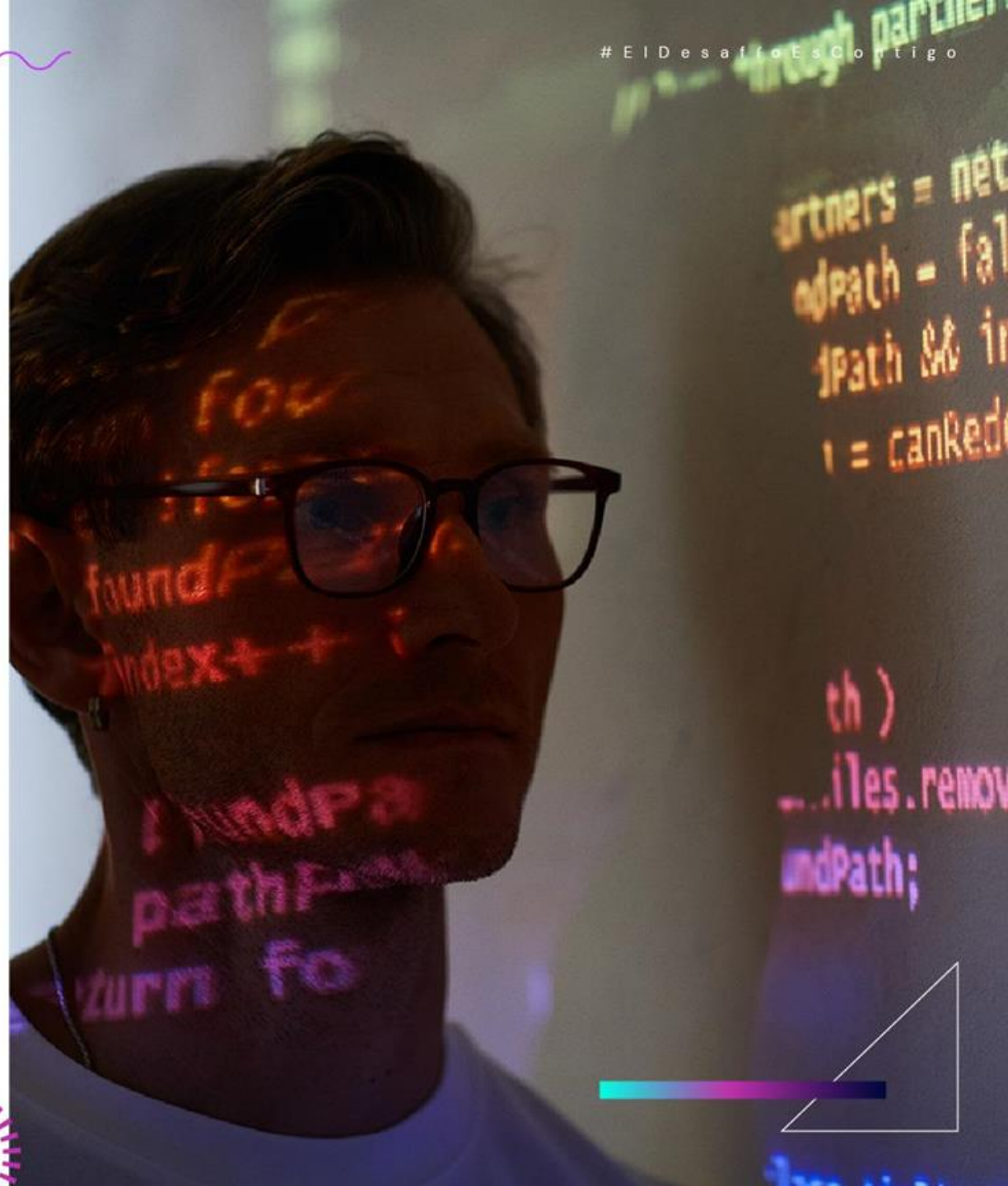
- El valor de una expresión depende únicamente de los valores de las expresiones que la constituyen (si es que existen) y estas sub-expresiones pueden reemplazarse libremente por otras que posean el mismo valor.



Diferencia

- Las sentencias se utilizan para realizar efectos secundarios, como operaciones de E/S o cambiar el estado de un proceso, mientras que las expresiones se utilizan para calcular valores.

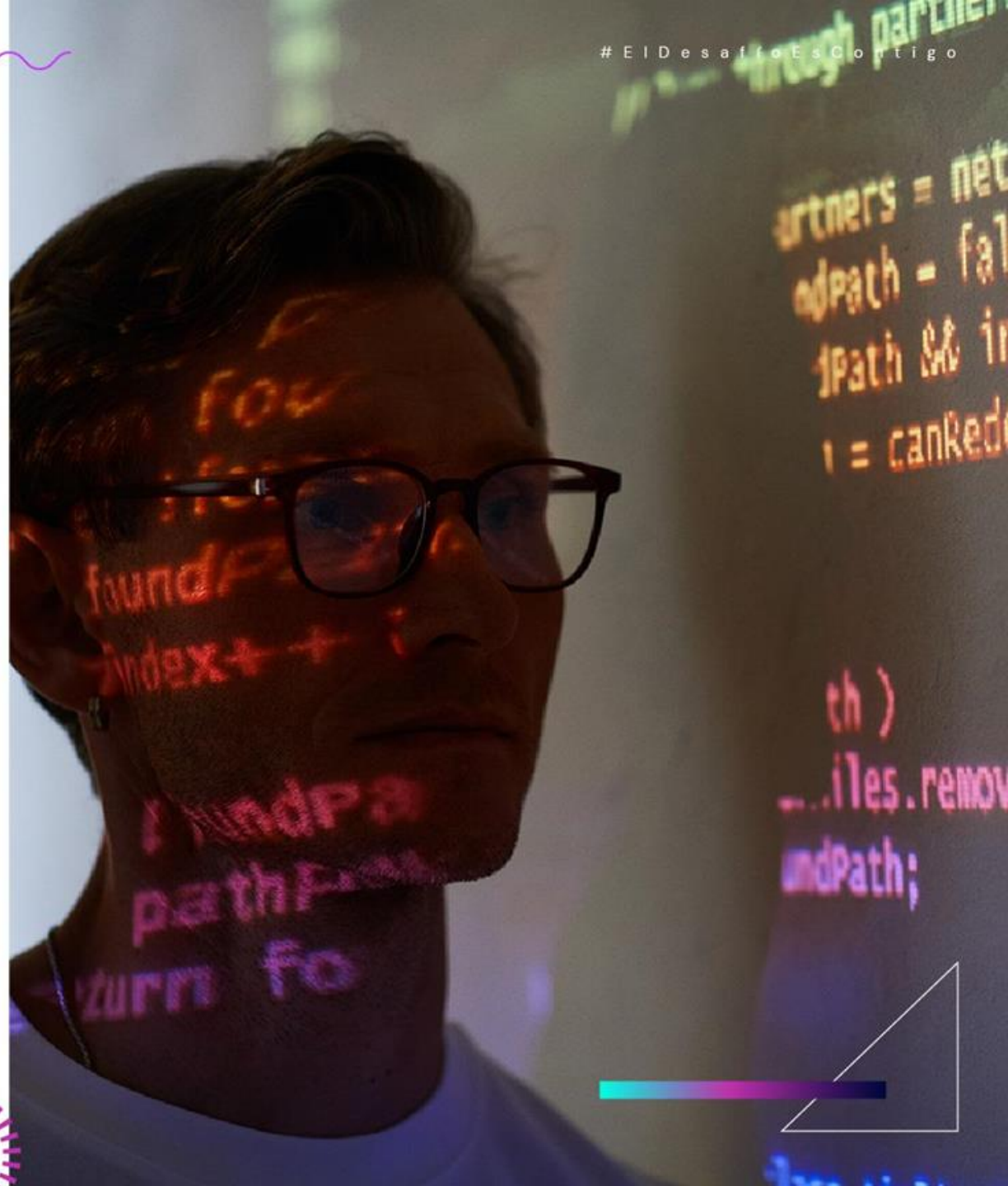
```elixir
if File.exists?("file.txt") do
  File.write("file.txt", "Hello, world!")
  :ok
else
  :error
end
```



Reducers

- El uso de Reducers son uno de los conceptos clave de la programación funcional y son el mejor ejemplo de ‘expresar’.

```
numbers = [2, 3, 4]
IO.inspect(numbers, label: "Numbers")
r = Enum.reduce(numbers, 0, fn(n, acc) ->
  IO.inspect(acc, label: "Accumulator")
  IO.inspect(n, label: "Element")
  IO.inspect(n + acc, label: "Product")
end)
IO.inspect(r, label: "Reduce")
```

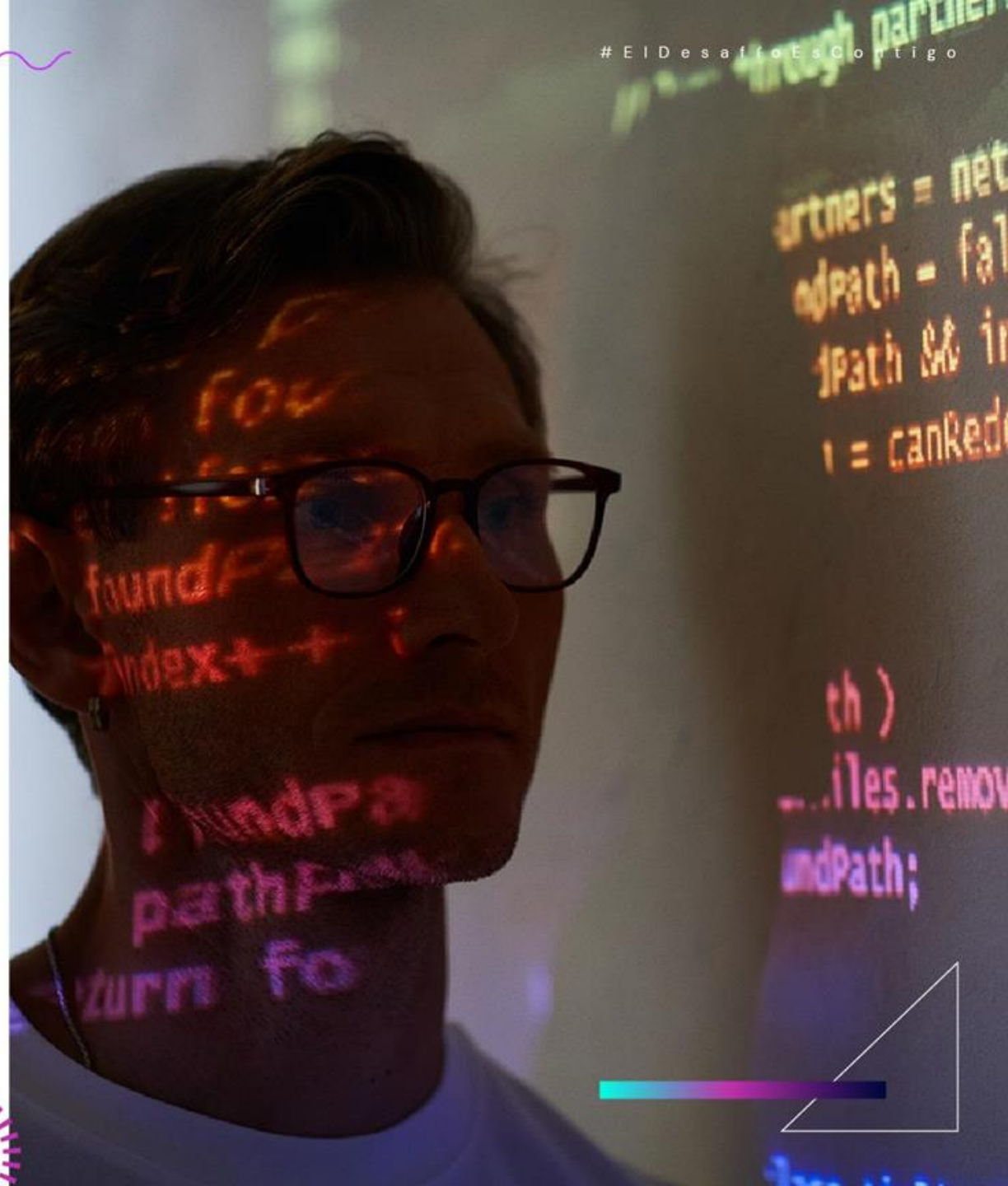


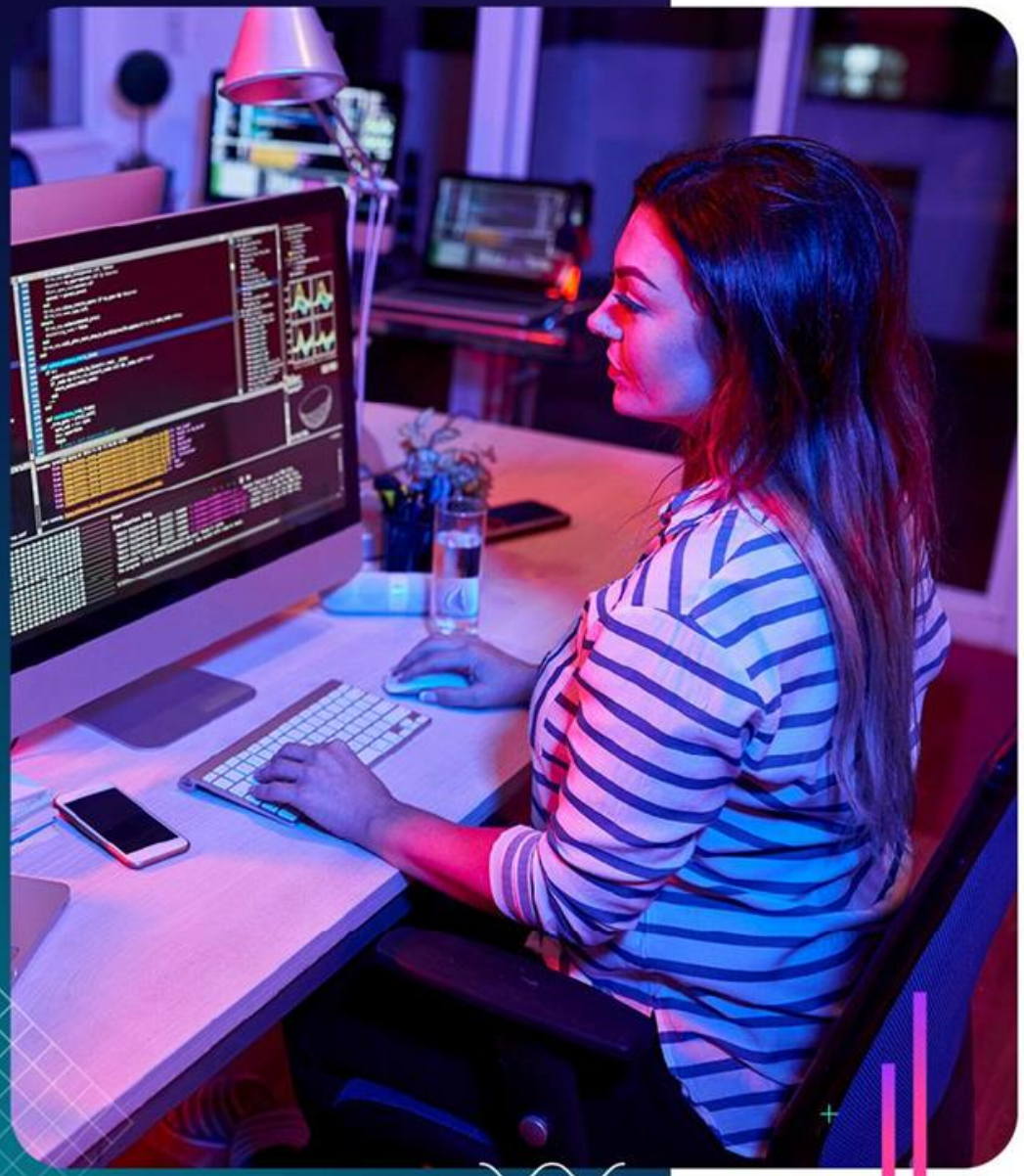
Programación funcional
Colecciones II

Keyword:

- Su traducción literal es ‘lista de palabras claves’.
- Se parecen mucho a los mapas y tienen una interfaz similar a la de los mapas, pero en realidad se implementan como listas de tuplas.
- Una palabra clave puede tener claves duplicadas, por lo que no es estrictamente un tipo de datos clave-valor.

```
iex> option_list = [size: 12, color: "red", color: "blue", style: :bold, style: :italic]
[size: 12, color: "red", color: "blue", style: :bold, style: :italic]
```





Características

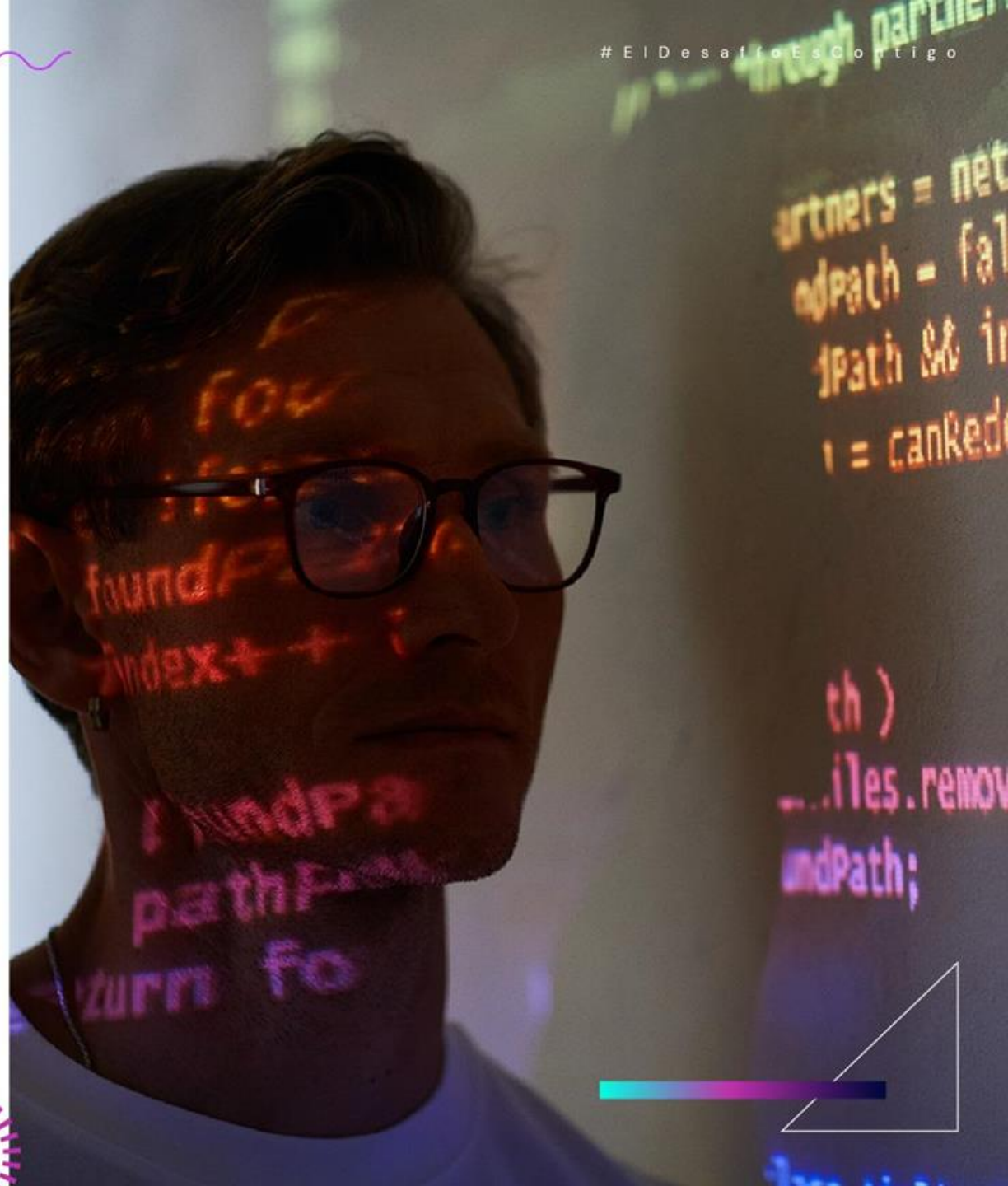
- Las operaciones de lista de palabras clave son más lentas que sus contrapartes de mapa.
- Se parecen mucho a los mapas y tienen una interfaz similar a la de los mapas, pero en realidad se implementan como listas de tuplas.
- Una palabra clave puede tener claves duplicadas, por lo que no es estrictamente un tipo de datos clave-valor.

Keyword y funciones:

- Cuando se pasan listas de palabras clave como último argumento de una función, se pueden omitir los corchetes alrededor de la lista de palabras clave.

```
String.split("1-0", "-", [trim: true, parts: 2])
```

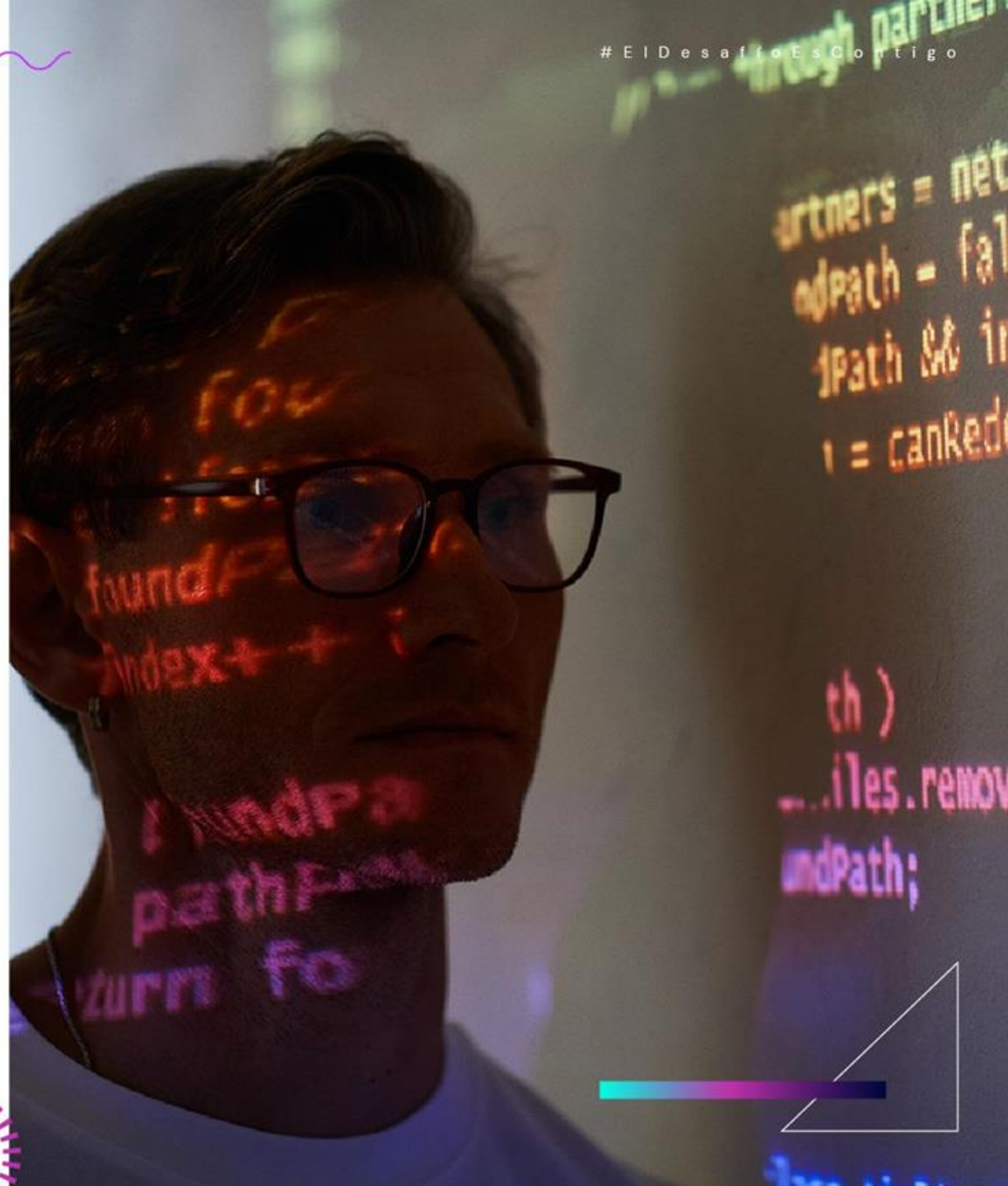
```
String.split("1-0", "-", trim: true, parts: 2)
```

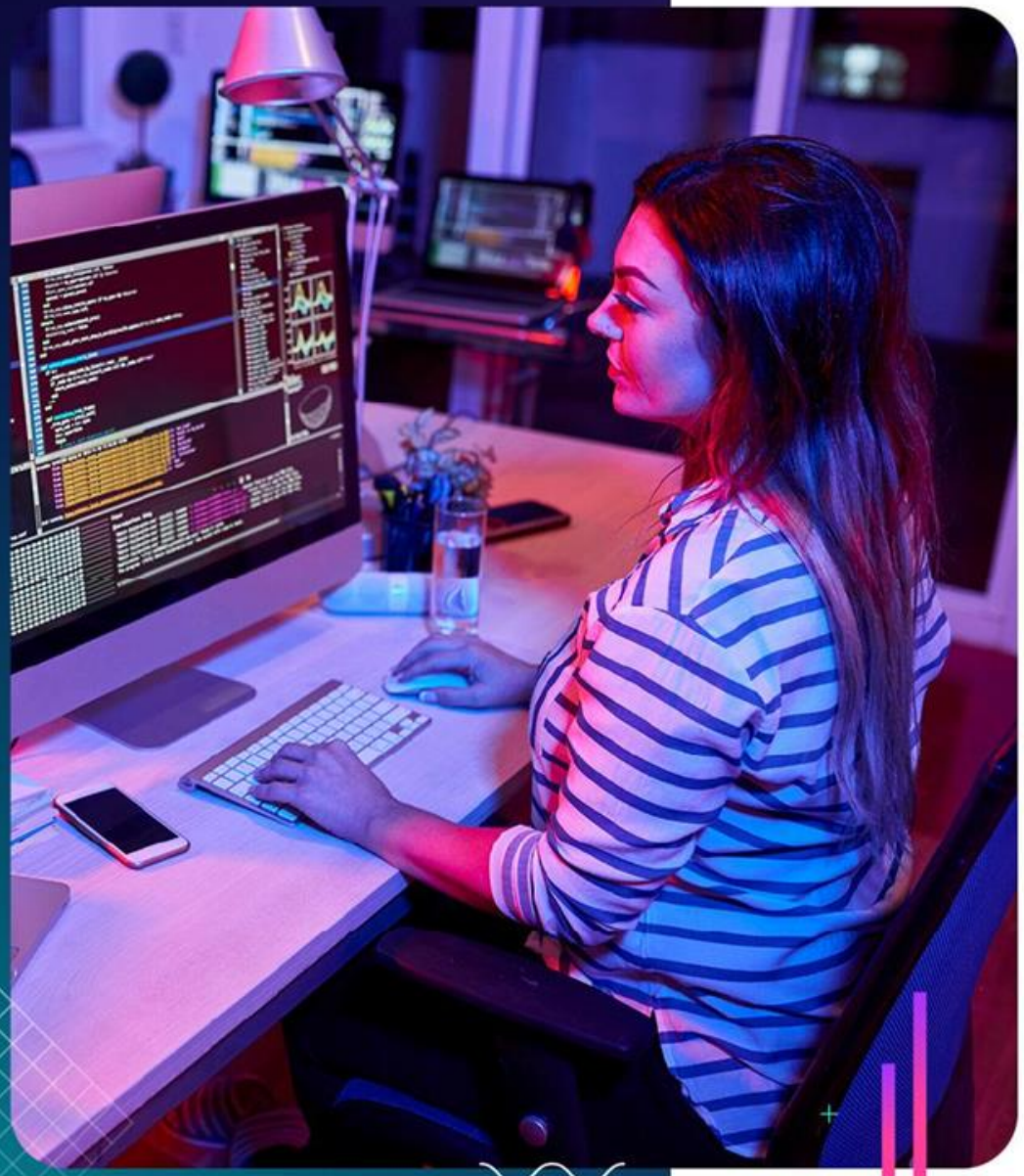


Tupla:

- Las tuplas están pensadas como contenedores de tamaño fijo para múltiples elementos.
- Se crean con la notación de { }

```
iex> {}
{}
iex> {1, :two, "three"}
{1, :two, "three"}
```





Características

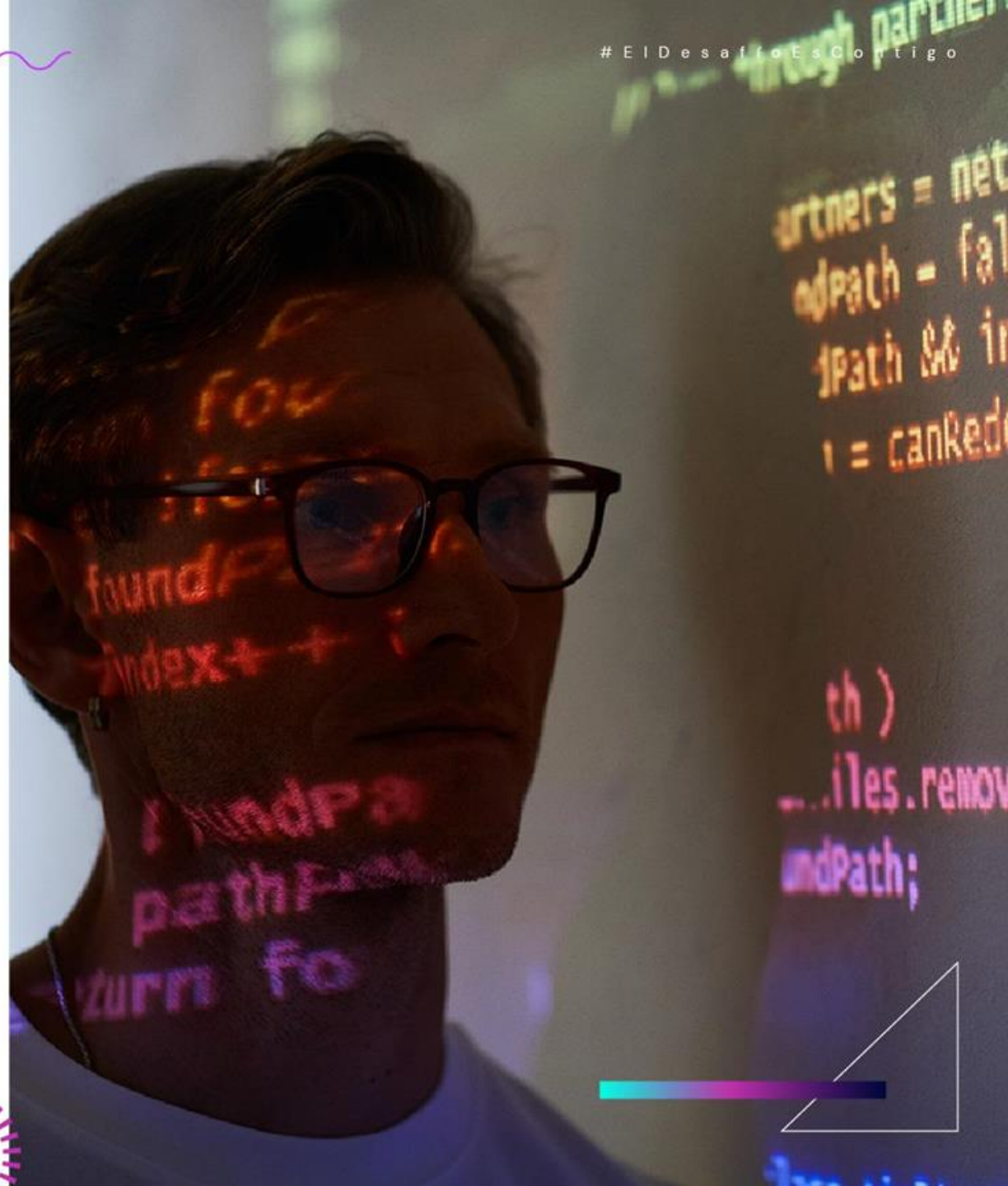
- Una tupla puede contener elementos de diferentes tipos
- A diferencia de las listas, almacenan elementos en un bloque contiguo de memoria.
- Se utilizan normalmente cuando una función tiene múltiples valores de retorno o para el manejo de errores.

Range:

- Representan una secuencia de cero, uno o muchos, enteros ascendentes o descendentes con una diferencia común llamada paso o ‘step’.

- Los rangos suelen tener dos usos en Elixir: como colección o para representar una porción de otra estructura de datos.

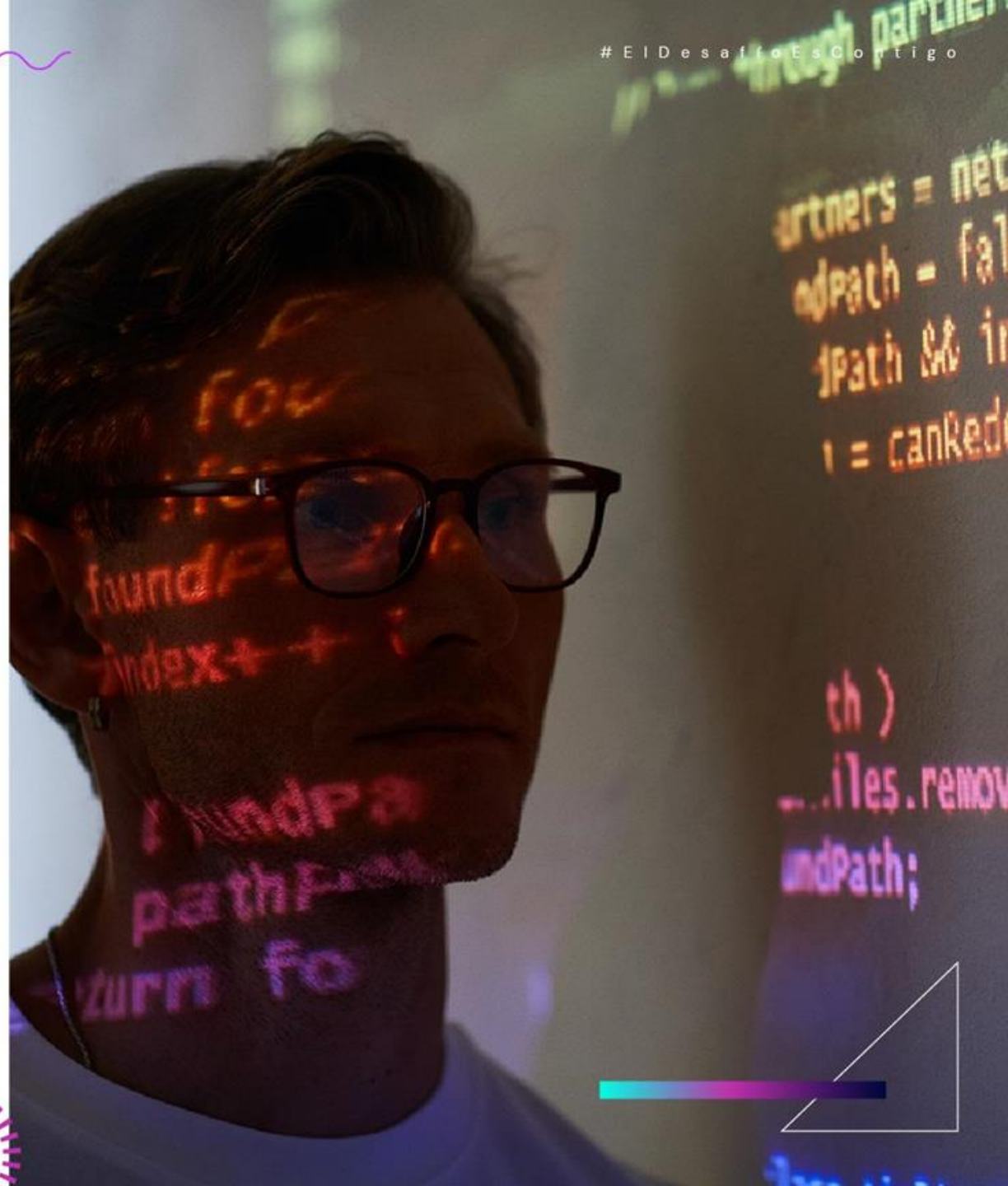
```
iex> 1 in 1..10
true
iex> 5 in 1..10
true
iex> 10 in 1..10
true
```



Range (Colección):

- Los rangos en Elixir son enumerables y por lo tanto pueden ser utilizados con el módulo Enum

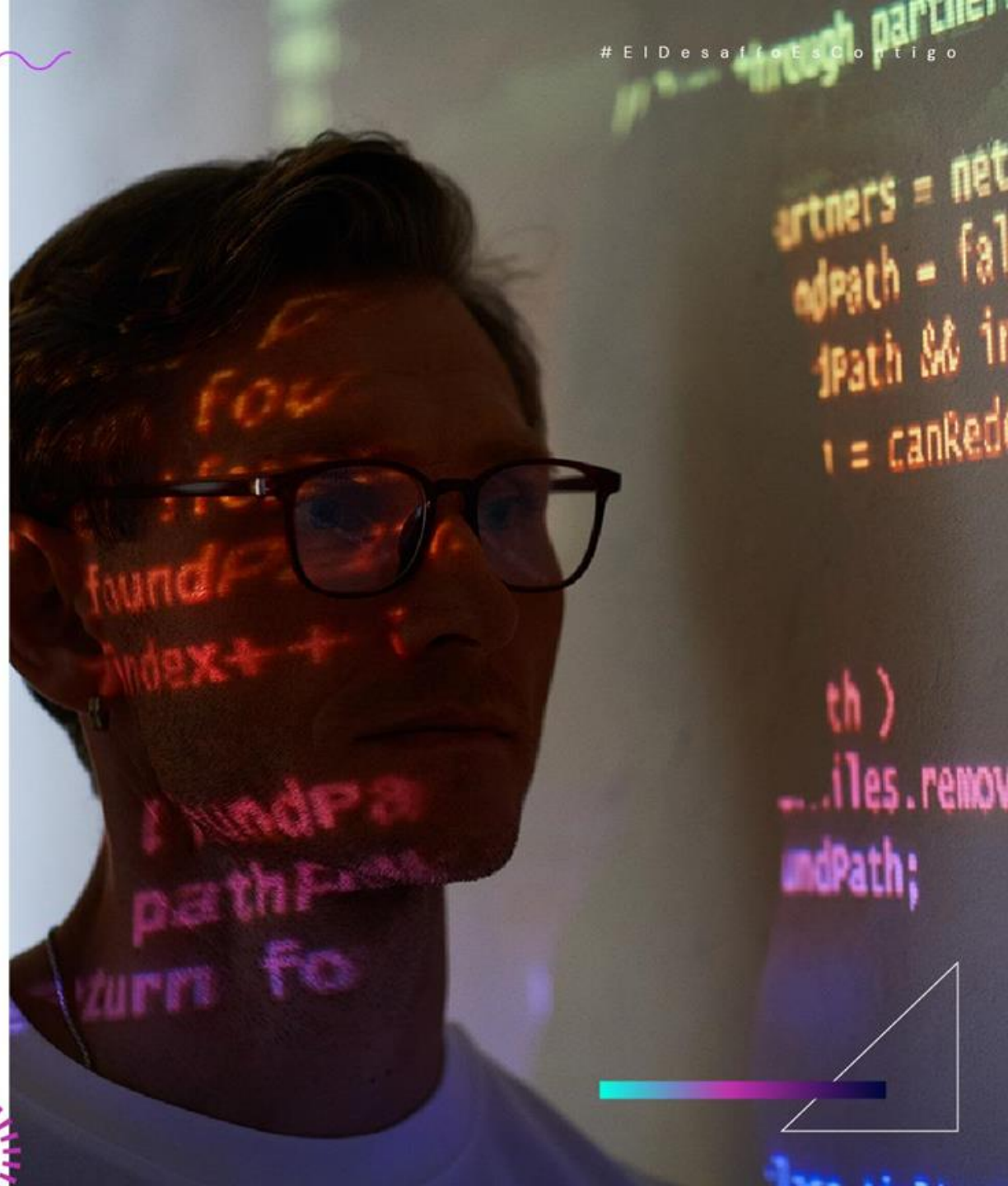
```
iex> Enum.to_list(1..3)
[1, 2, 3]
iex> Enum.to_list(3..1//-1)
[3, 2, 1]
iex> Enum.to_list(1..5//2)
[1, 3, 5]
```

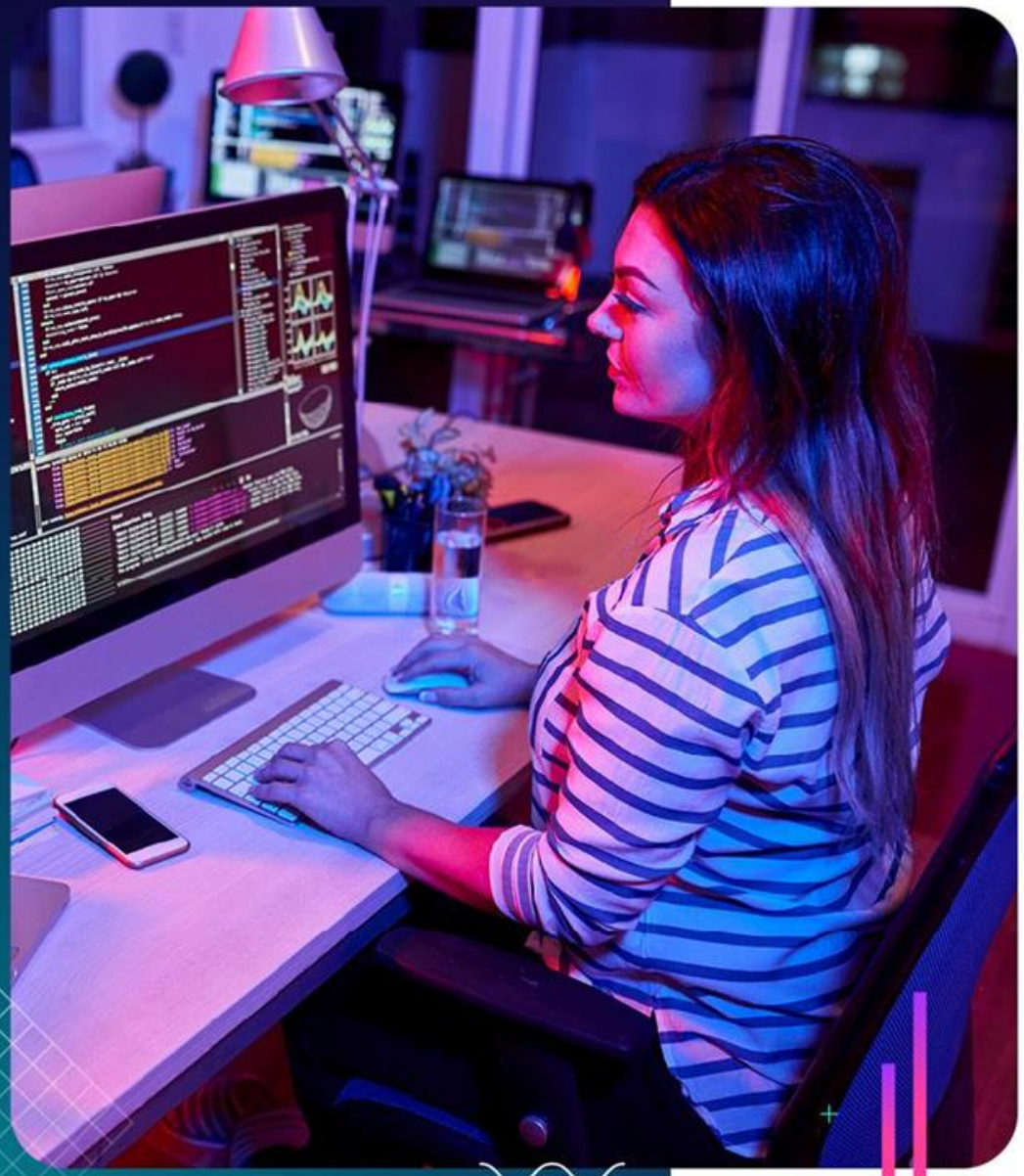


Range (Slicing):

- Los rangos también se utilizan con frecuencia para hacer slicing en colecciones.

```
iex> String.slice("elixir", 1..4)
"lixi"
iex> Enum.slice([0, 1, 2, 3, 4, 5], 1..4)
[1, 2, 3, 4]
```





Características

- Los rangos implementan el protocolo Enumerable con versiones eficientes en memoria para cualquier llamado en otras funciones.

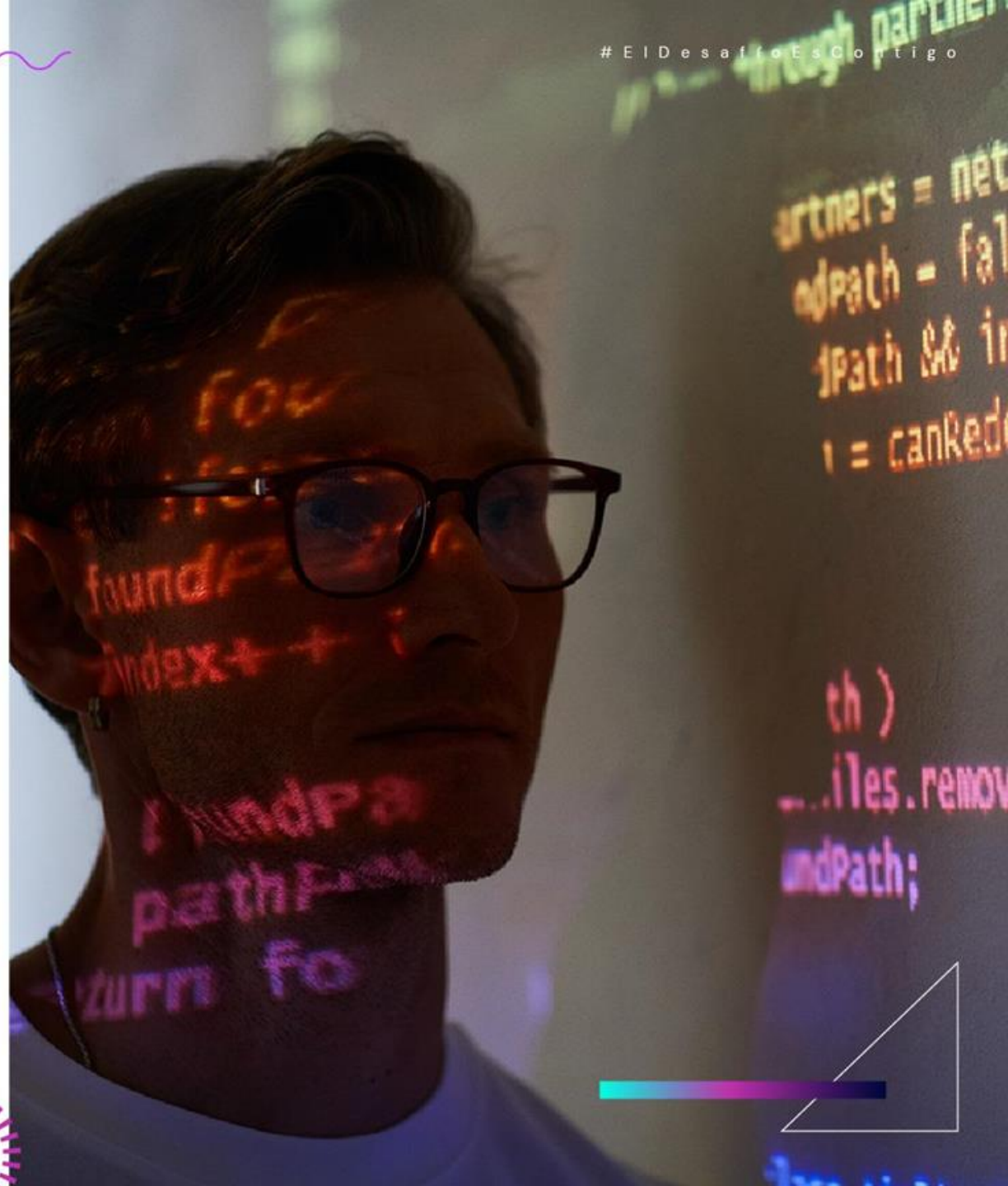
- Podemos crear rangos ascendentes y descendentes e incluso aplicar estructura.

```
iex> range = 1..9//2
1..9//2
iex> first..last//step = range
iex> first
1
iex> last
9
iex> step
2
iex> range.step
2
```

Excepciones

- Son una forma de manejar errores o eventos inesperados que ocurren durante la ejecución del programa.
- Las excepciones le permiten manejar errores con gracia y proporcionar información útil para el usuario, en lugar de permitir que el programa se bloquee o salga inesperadamente.

```
try do
  # some code that might raise an exception
catch
  RuntimeError -> IO.puts("Runtime error occurred")
  ArgumentError -> IO.puts("Invalid argument")
after
  IO.puts("This will always run")
end
```



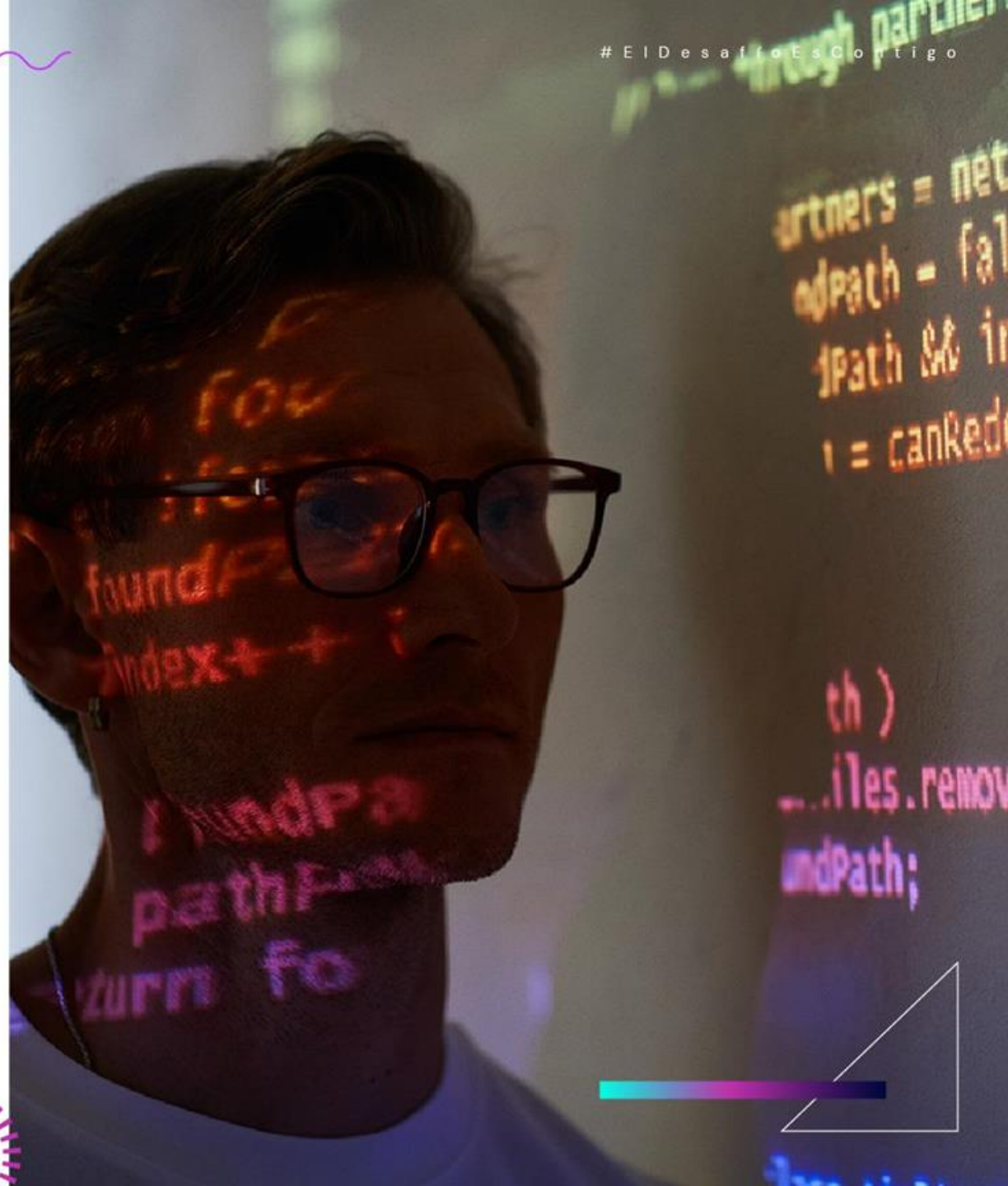
Puntos clave

1. El tipo de excepción más común es RuntimeError, que se utiliza para errores generales de ejecución.

2. Las excepciones pueden plantearse utilizando la función raise/1, que toma una estructura de excepción como argumento

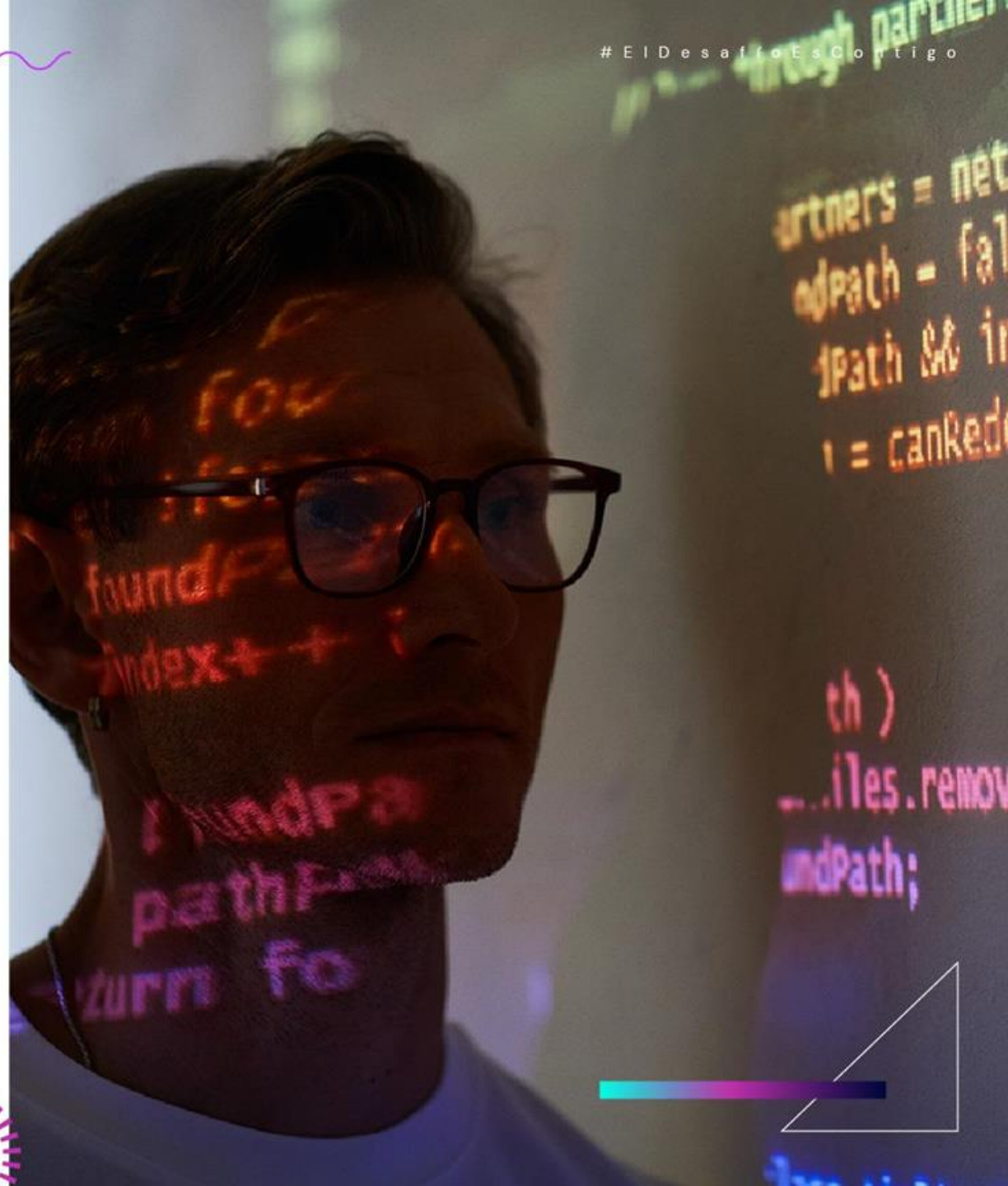
3. Puedes definir tus propios structs de excepción que implementen el protocolo Exception.

4. Try-catch-after vs Try-rescue-after



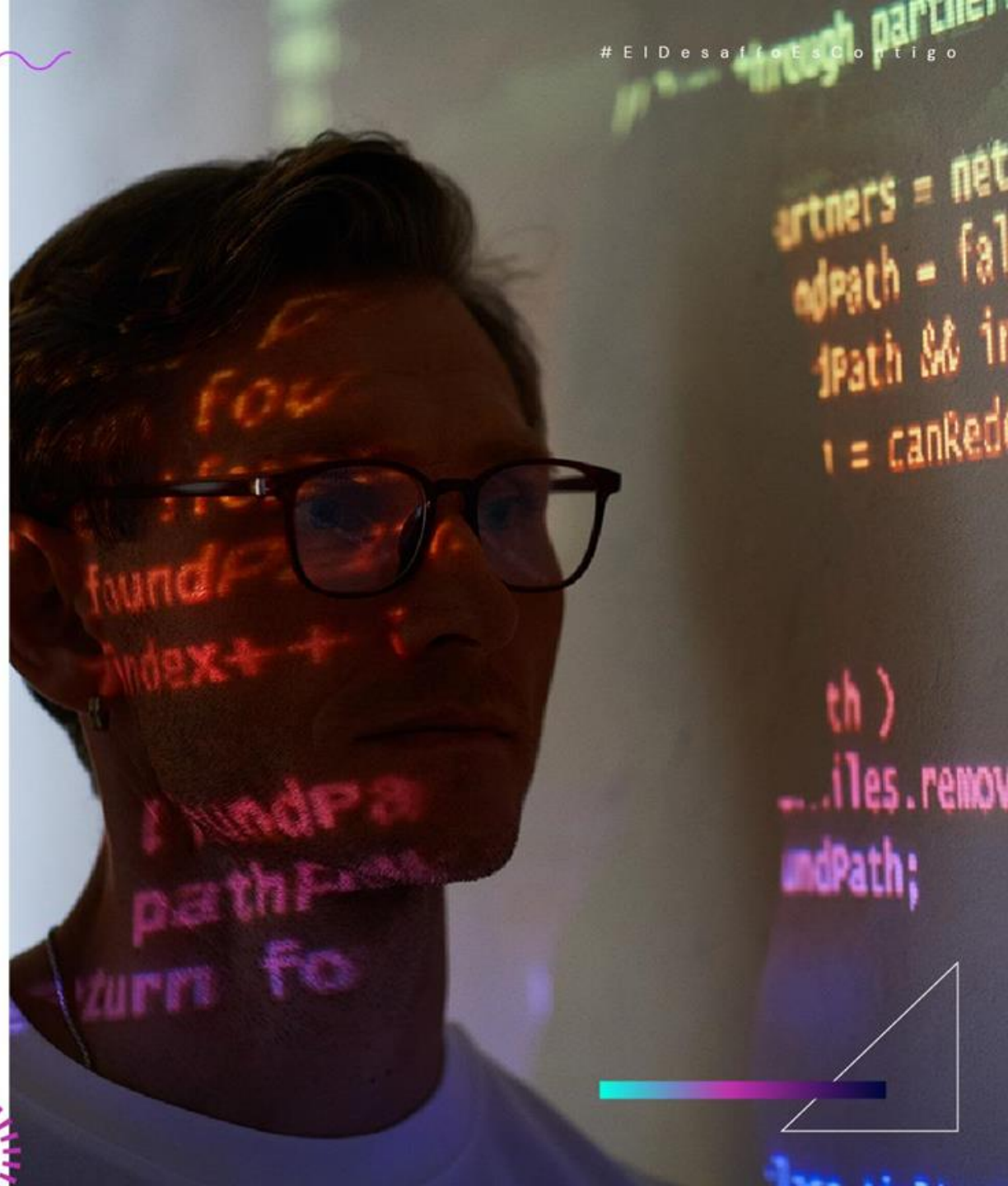
Try-catch

- Try-catch se utiliza para atrapar errores en tiempo de ejecución
- Se utiliza para errores no fatales, como cuando se desea manejar un error específico y luego continuar ejecutando el programa.
- La cláusula catch en try-catch debe ser capaz de manejar la excepción exacta que se lanza, de lo contrario la excepción se propagará por la pila de llamadas.



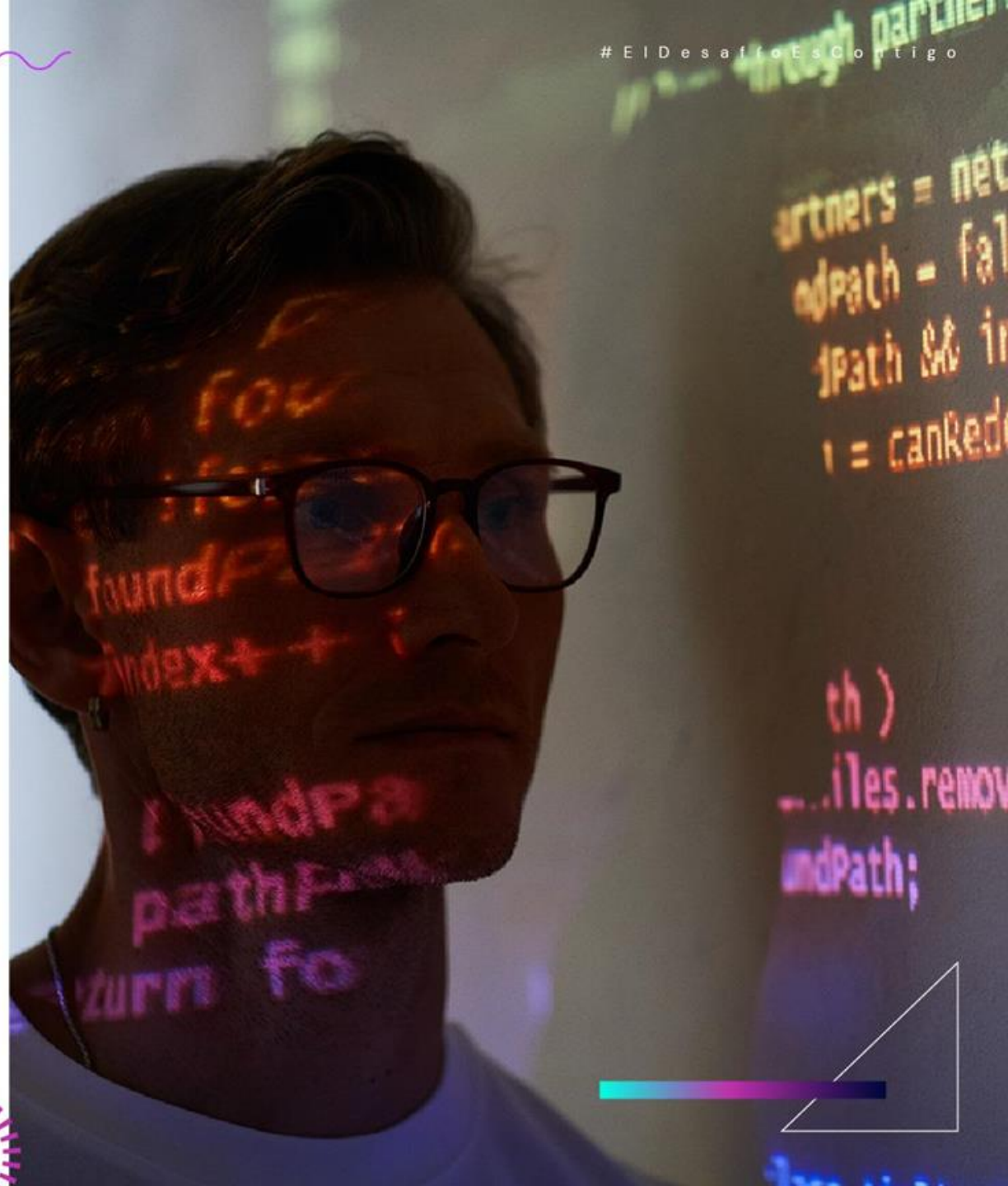
Try-rescue

- try-rescue se utiliza para manejar errores que se plantean explícitamente con raise/1 o raise/2.
- Se utiliza a menudo para manejar errores inesperados
- Estos errores suelen ser más graves e indican que el programa no puede continuar ejecutándose como se esperaba.



Ejemplo

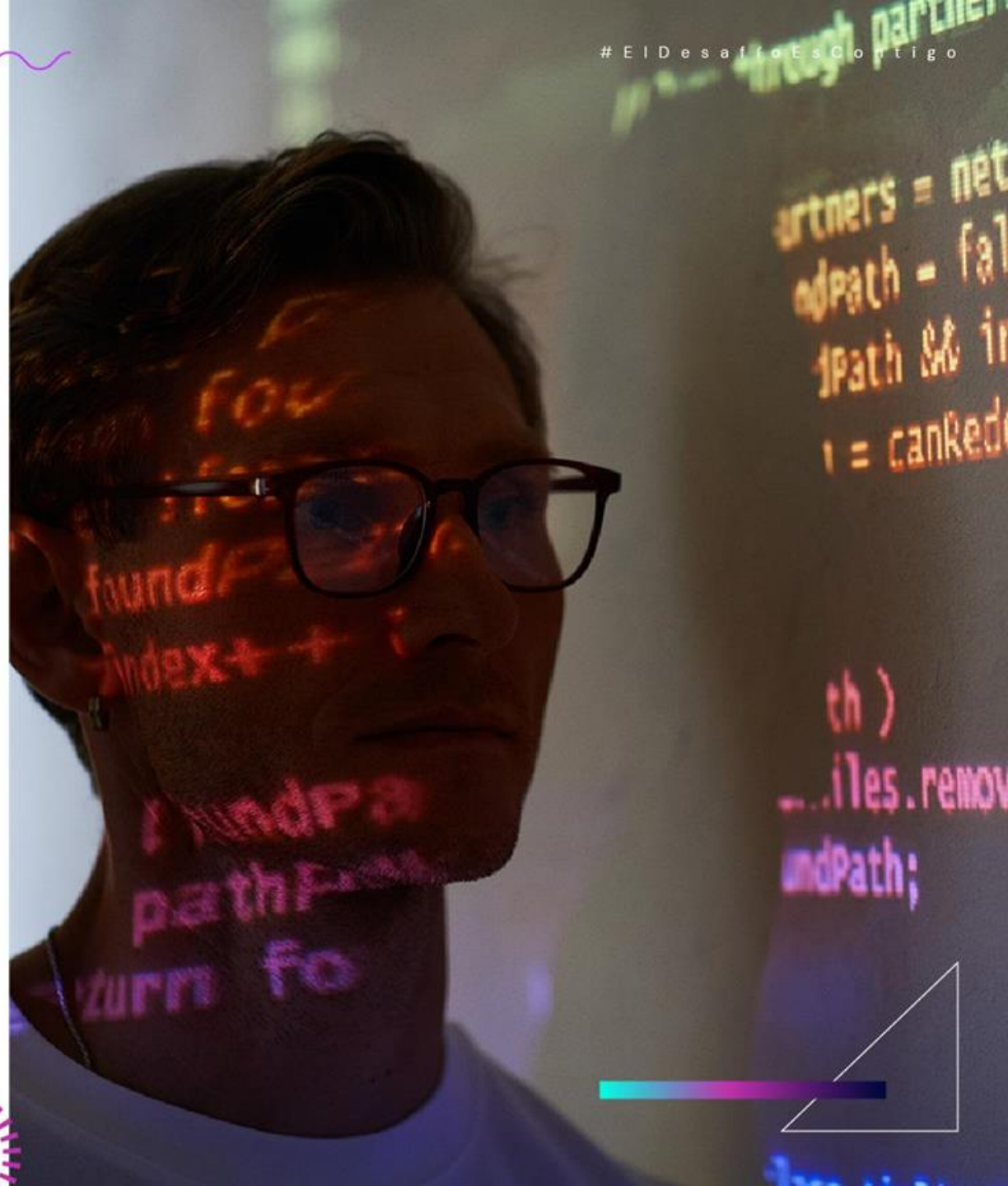
```elixir
defmodule Example do
  def divide(a, b) do
    if b == 0 do
      raise ArgumentError, "Cannot divide by zero"
    else
      a / b
    end
  end

  def process_input(input) do
    try do
      value = Integer.parse(input)
      divide(10, value)
    catch
      ArgumentError -> IO.puts("Invalid input: #{input}")
      _ -> IO.puts("An error occurred")
    end
  end
end

# Test the process_input function with different inputs
Example.process_input("0") # Should output "Invalid input: 0"
Example.process_input("2") # Should output "5"
Example.process_input("abc") # Should output "An error occurred"
```



Programación funcional + Excepciones

Si, pero no.

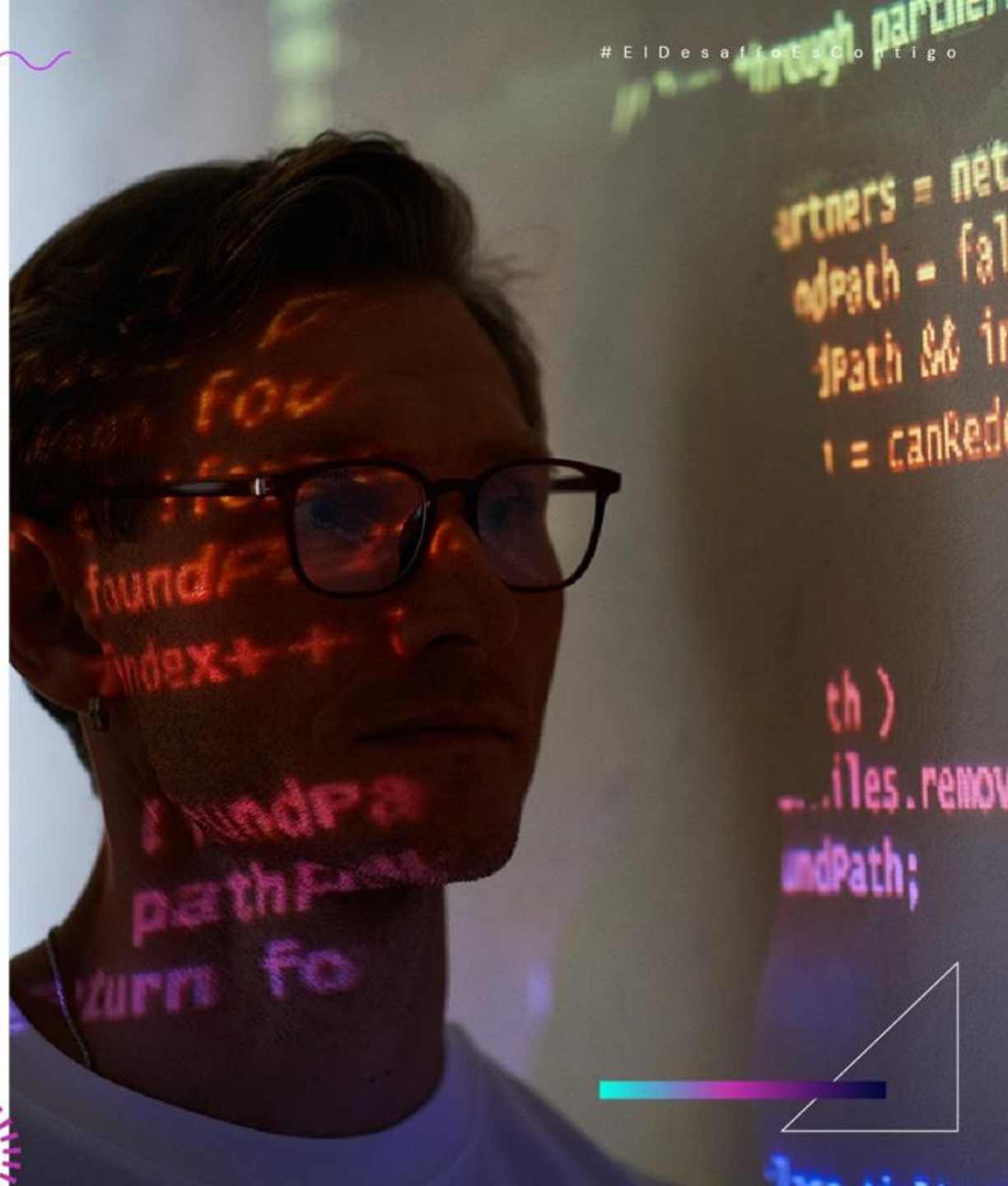
- Las excepciones proporcionan una manera de manejar errores y condiciones excepcionales en Elixir de una manera que es a la vez concisa y expresiva.
- Cuando se produce una excepción, Elixir desenrollará la pila de llamadas hasta que encuentre un bloque de rescate que pueda manejar la excepción.
- Esto puede ser especialmente útil cuando se trata de situaciones inesperadas



¿Entonces?

- Sigue siendo importante esforzarse por conseguir un código lo más robusto y libre de errores posible, escribiendo funciones puras, utilizando comprobación de tipos y realizando pruebas exhaustivas.

Sin embargo, cuando se producen errores, las excepciones pueden ser una herramienta valiosa para gestionarlos.



Conclusiones

- Las Keywords en Elixir son un tipo de estructura de datos que permiten pasar argumentos con nombre a las funciones, proporcionando una sintaxis más legible y expresiva.

- Las excepciones nos permiten manejar situaciones inesperadas que pueden surgir durante la ejecución del programa, como errores o entradas no válidas.

- Se recomienda utilizar las excepciones con moderación y sólo en los casos en que sean realmente necesarias. A menudo es mejor utilizar técnicas de programación funcional.

- Las tuplas son una estructura de datos fundamental en Elixir y se utilizan a menudo para devolver múltiples valores de una función.

[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