# Why Exceptions Are Just Sophisticated GoTos

# … and How to Move Beyond

PyConDE / PyData 2025, April 23rd

**Florian Wilhelm**

inovex

# Dr. Florian Wilhelm

inovex • HEAD OF DATA SCIENCE

FlorianWilhelm

FlorianWilhelm.info

florian.wilhelm@inovex.de

- Mathematical Modelling
- Modern Data Warehousing & Analytics
- Personalisation & RecSys
- Uncertainty Quantification & Causality
- Python Data Stack
- OSS Contributor & Creator of PyScaffold

inovex

# WE ARE INOVEX:

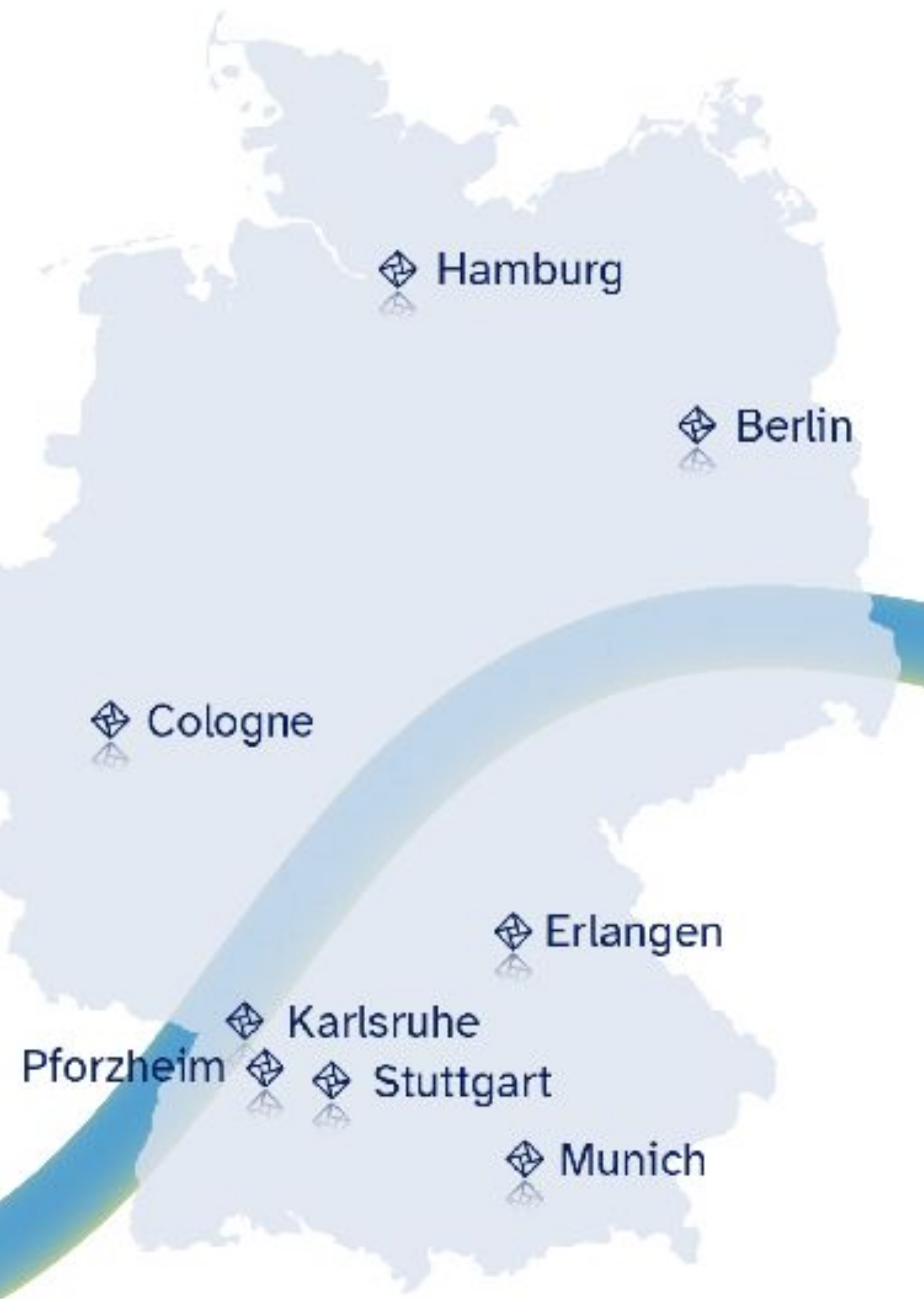
- IT Project Center
- Innovation & Excellence
- Wide Range of Services

More Than
20 inovex Attendees on Site
6 Talks | 1 Special Workshop for Kids
250 Python Users
500 Tech Heads Overall

WE ARE HIRING!
Data | Cloud | Backend | Frontend

inovex

INNOVATE. INTEGRATE. EXCEED.



**SOFTWARE · DATA & AI · INFRASTRUCTURE**

inovex.de

## Agenda

1. History of GoTo
2. Why Exceptions Exist and What They Are
3. The Evolution Toward Result Types
4. Using Result Types in Python
5. Conclusion

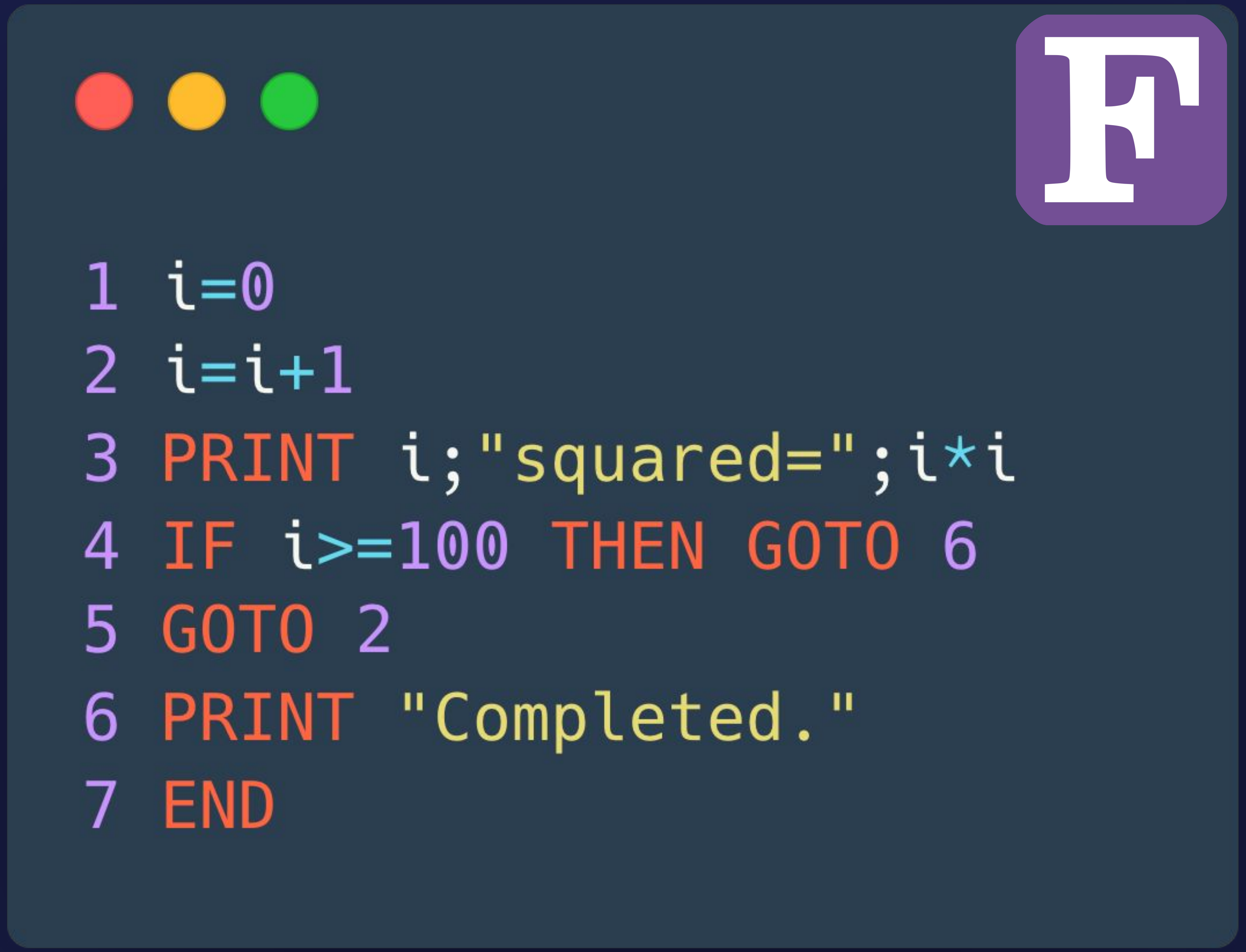
inovex

# History of GoTo

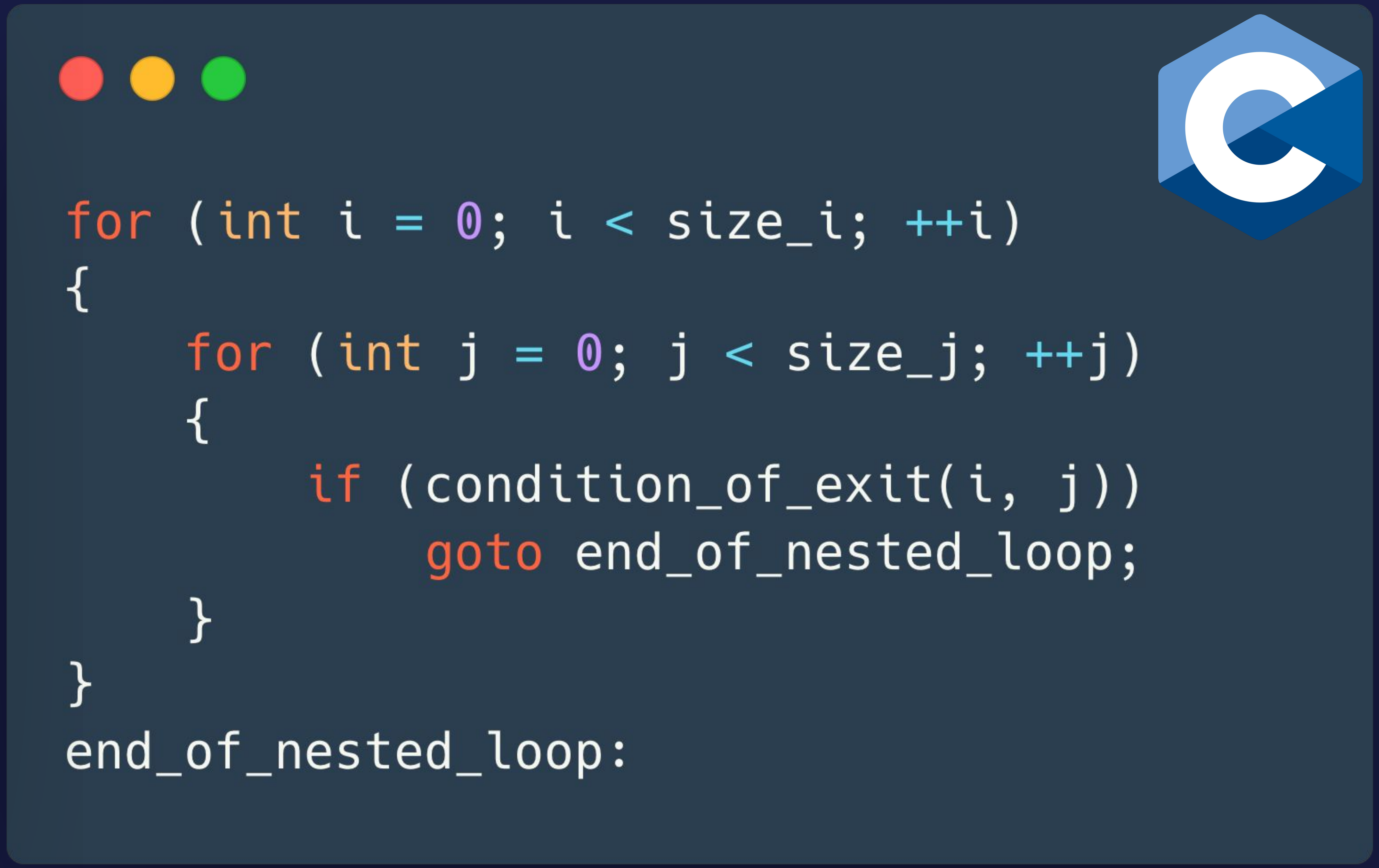
inovex

# GoTo in Fortran & C

GoTo is a jump to a label, i.e. one-way transfer of control to another line of code.



```
1 i=0
2 i=i+1
3 PRINT i;"squared=";i*i
4 IF i>=100 THEN GOTO 6
5 GOTO 2
6 PRINT "Completed."
7 END
```

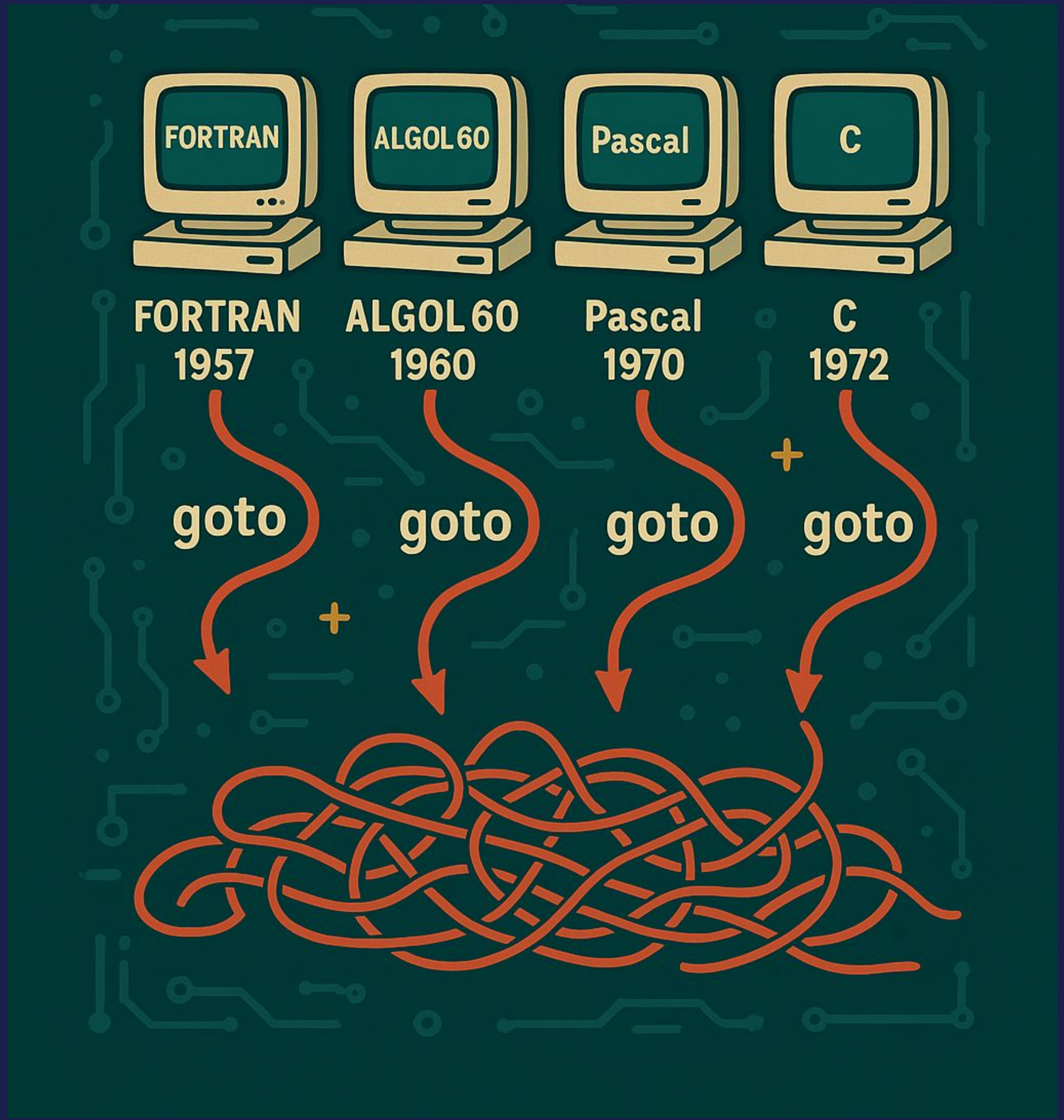


```
for (int i = 0; i < size_i; ++i)
{
    for (int j = 0; j < size_j; ++j)
    {
        if (condition_of_exit(i, j))
            goto end_of_nested_loop;
    }
}
end_of_nested_loop:
```

inovex

## FORmula TRANslation or just FORTRAN

- Fortran introduced GoTo and If statements in 1957
- Applications of GoTo:
  - to skip code
  - to loop over code
  - to break out of loops
  - for error handling



The History, Controversy, and Evolution of the Goto Statement by Andru Luvisi, 2008

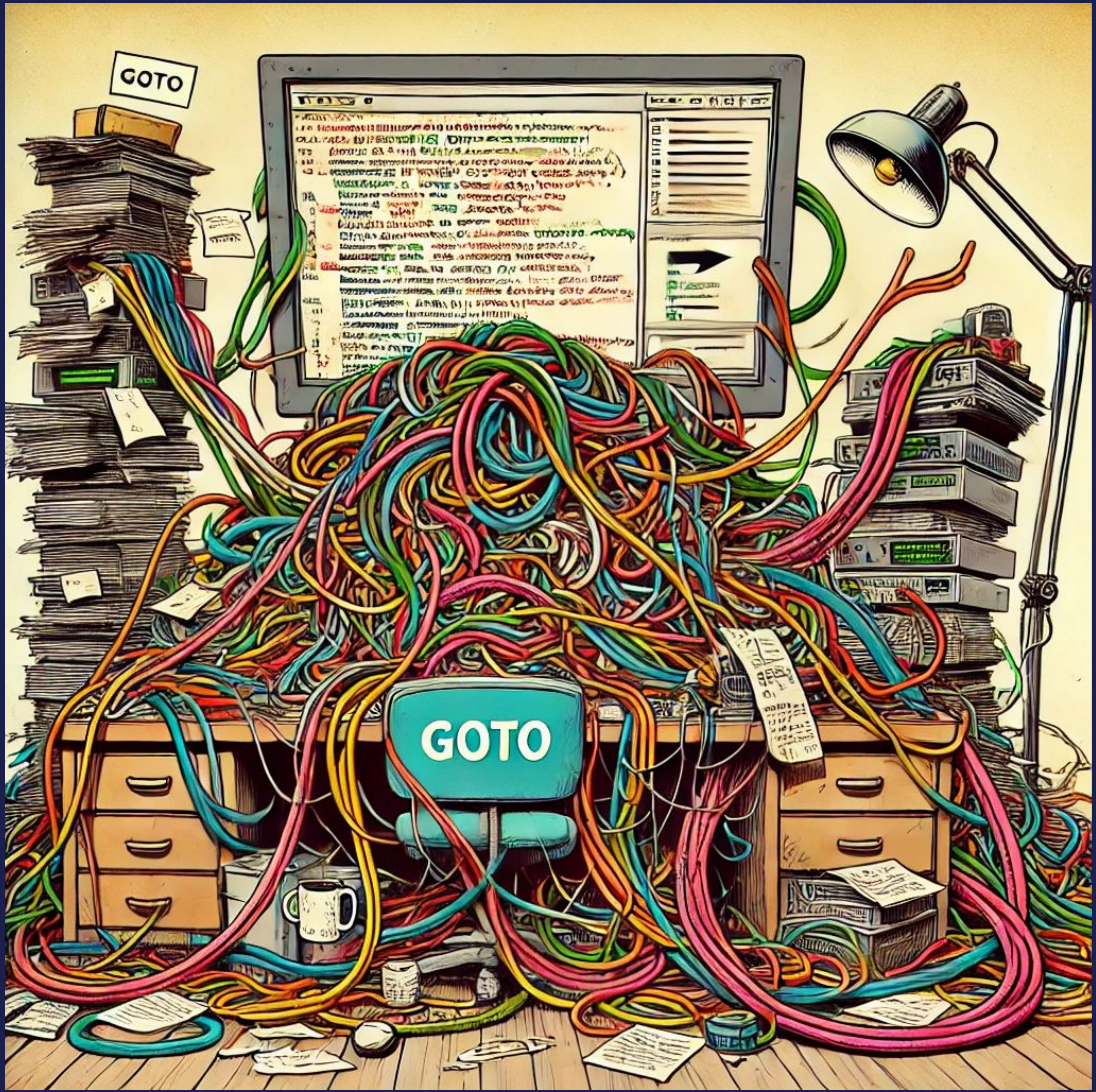
inovex

History of GoTo

# Spaghetti Code (1977)

Macaroni is Better than Spaghetti!
- Guy Lewis Steele, Jr.

https://dl.acm.org/doi/10.1145/800228.806933



inovex

History of GoTo

## Downsides of Goto

- Hard to understand
- Hard to follow the control flow (Spaghetti code)
- Extremely hard to debug

History of GoTo

## Structured Programming (~1960 with Algol60)

Structured programming to

- improve the clarity, quality, and development time of a computer program
- by using **structured control flow** like if/then/else, while/for-loop, block structures, e.g. begin/end, {} (or indentation), and subroutines.

Structured program theorem - Böhm-Jacopini (1966)

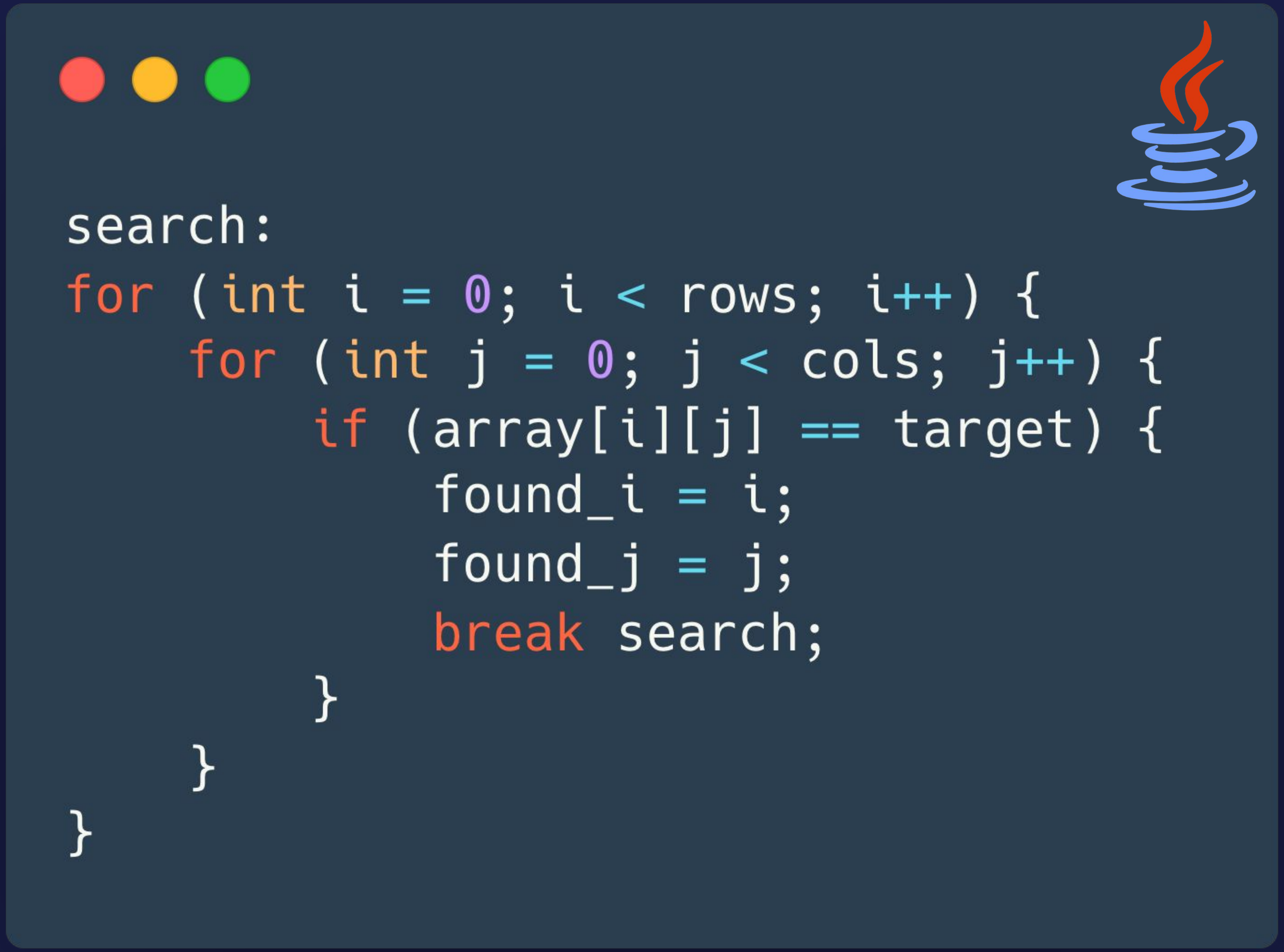
go to statement considered harmful! - Dijkstra (1968)

https://en.wikipedia.org/wiki/Structured_programming
go to statement considered harmful by Dijkstra, 1968

History of GoTo

# All non-trivial abstractions are leaky!

Knuth demonstrated that in certain cases, eliminating goto statements without introducing multi-level breaks or similar constructs can lead to less efficient or more complex code.



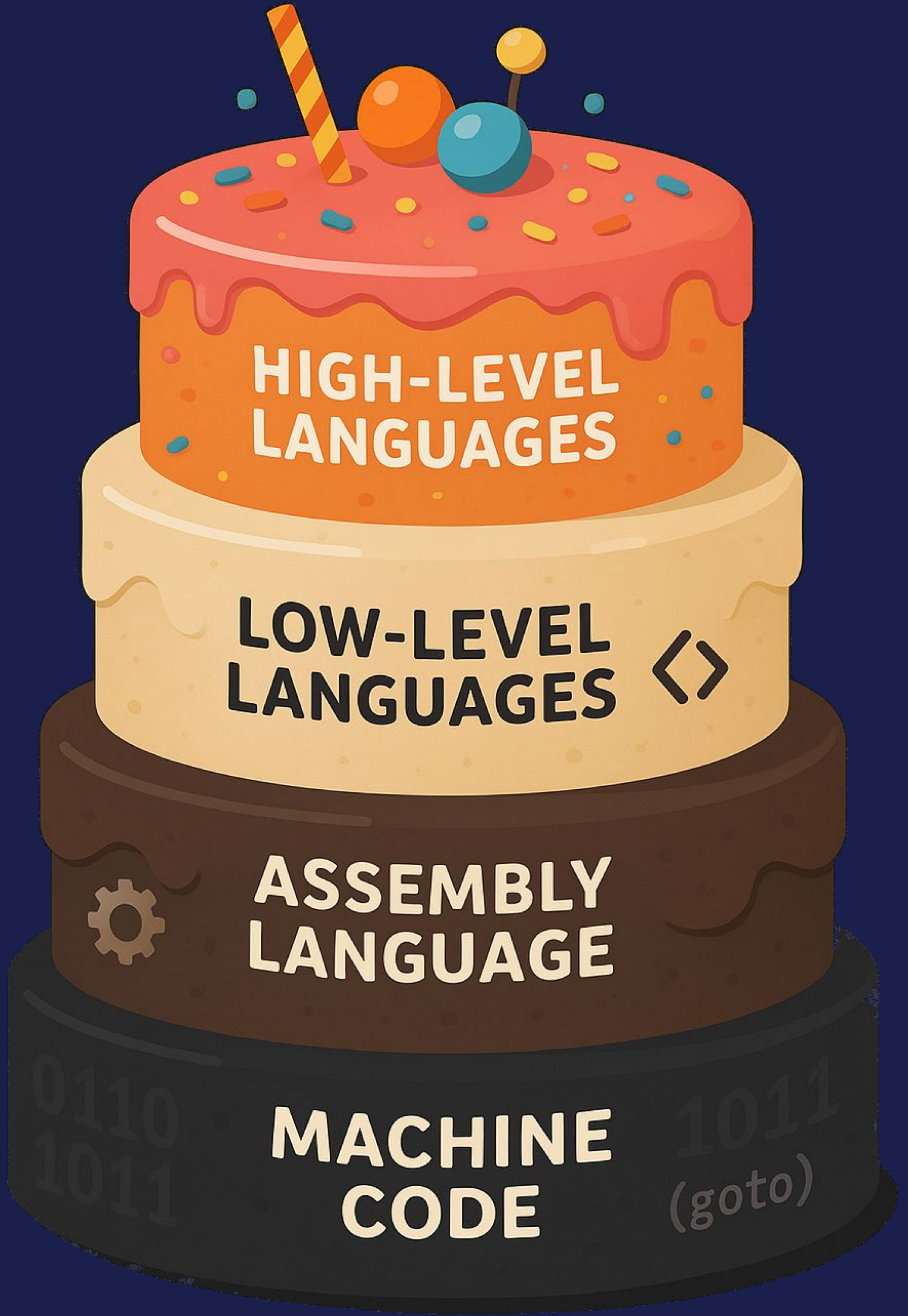
```
search:
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        if (array[i][j] == target) {
            found_i = i;
            found_j = j;
            break search;
        }
    }
}
```

Structured Programming with go to Statements by Donald E. Knuth, 1974

History of GoTo

# Layer Cake

GoTo is what the CPU does.
We abstract it to think better!



inovex

History of GoTo

# PEP 3136 – Labeled break and continue for Python 3.1 (2007)

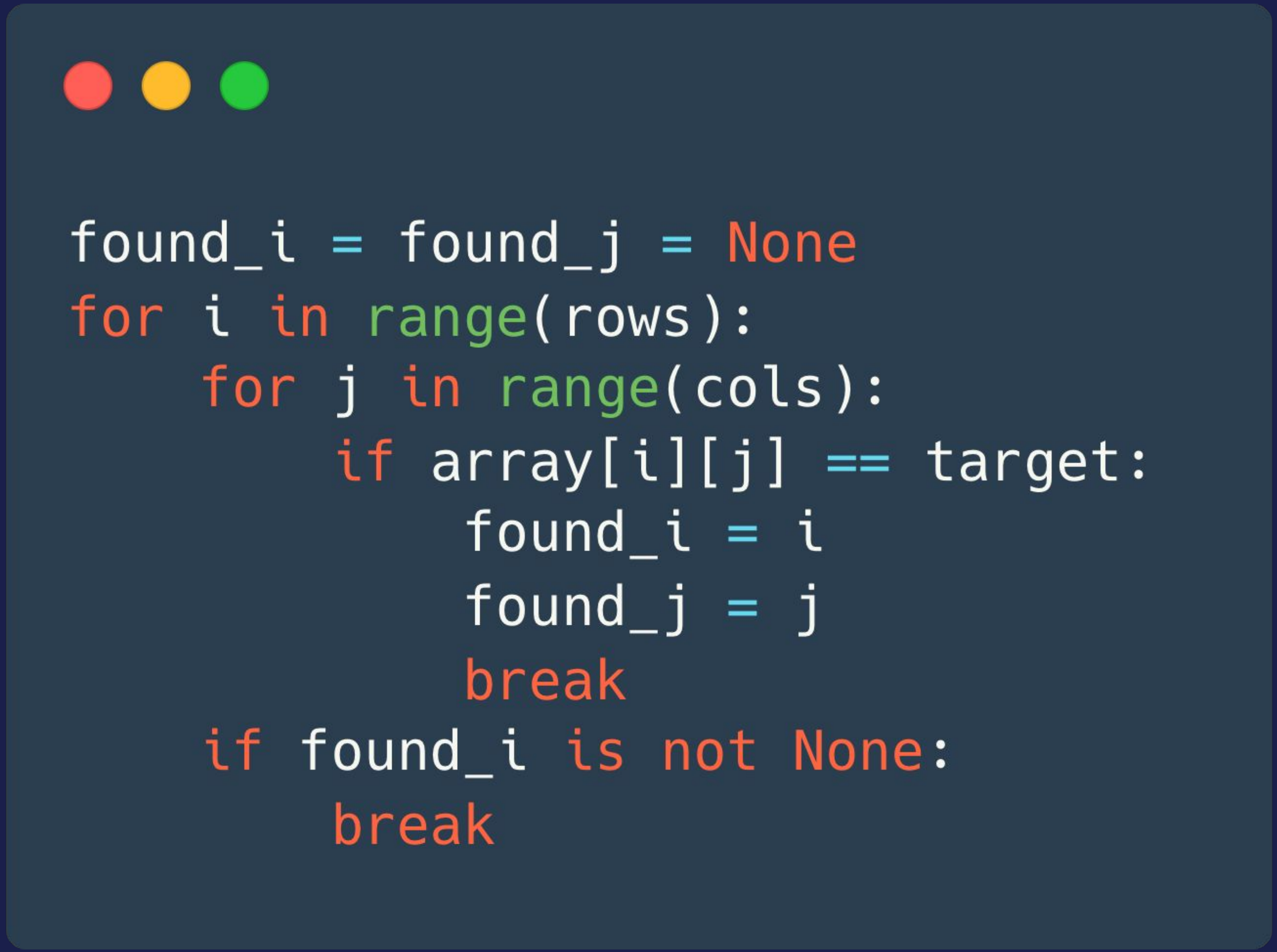
https://peps.python.org/pep-3136/

inovex

History of GoTo

## How about Python?

With a state

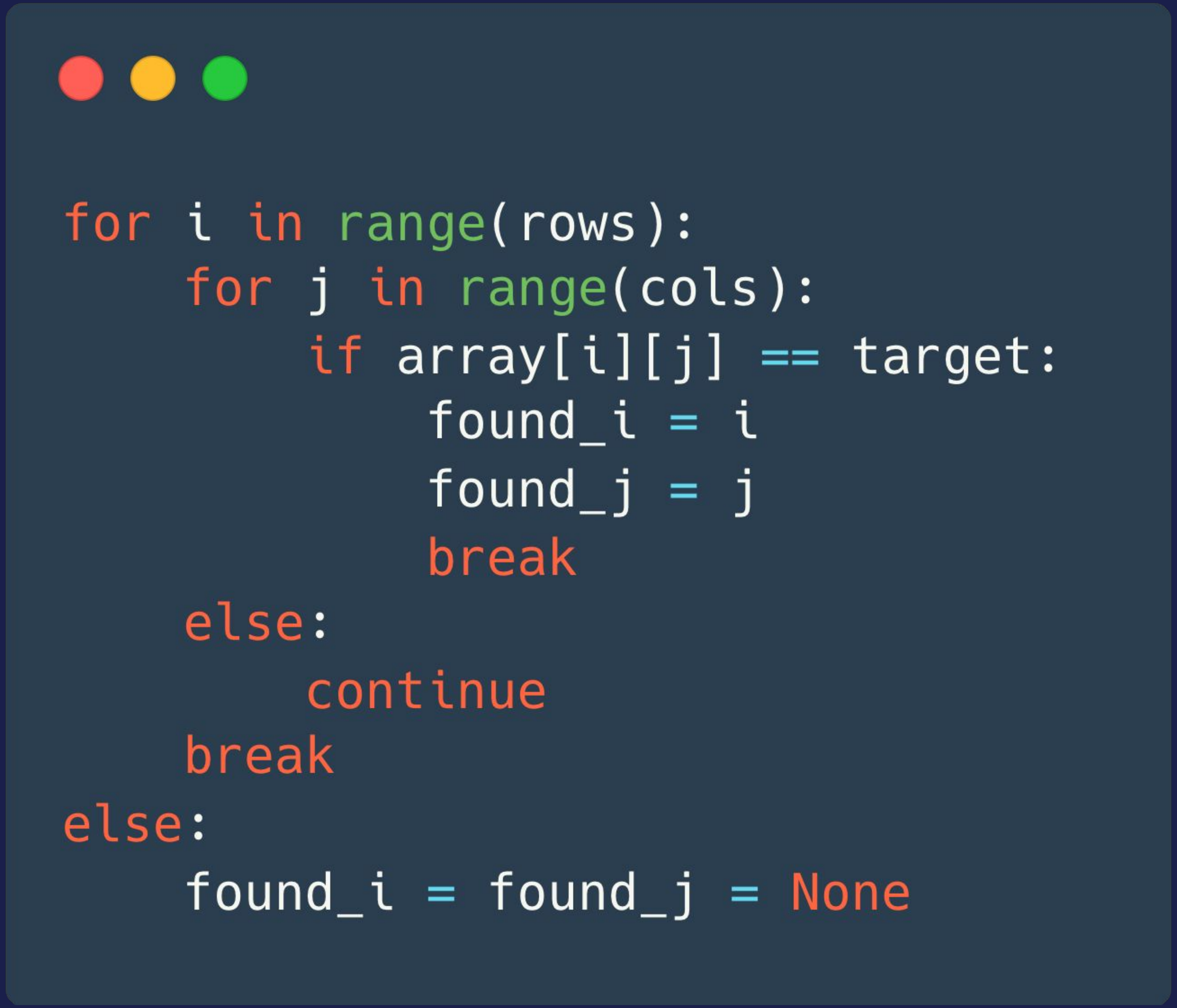


```python
found_i = found_j = None
for i in range(rows):
    for j in range(cols):
        if array[i][j] == target:
            found_i = i
            found_j = j
            break
    if found_i is not None:
        break
```

https://stackoverflow.com/questions/653509/breaking-out-of-nested-loops

## How about Python?

With else: of for-loop

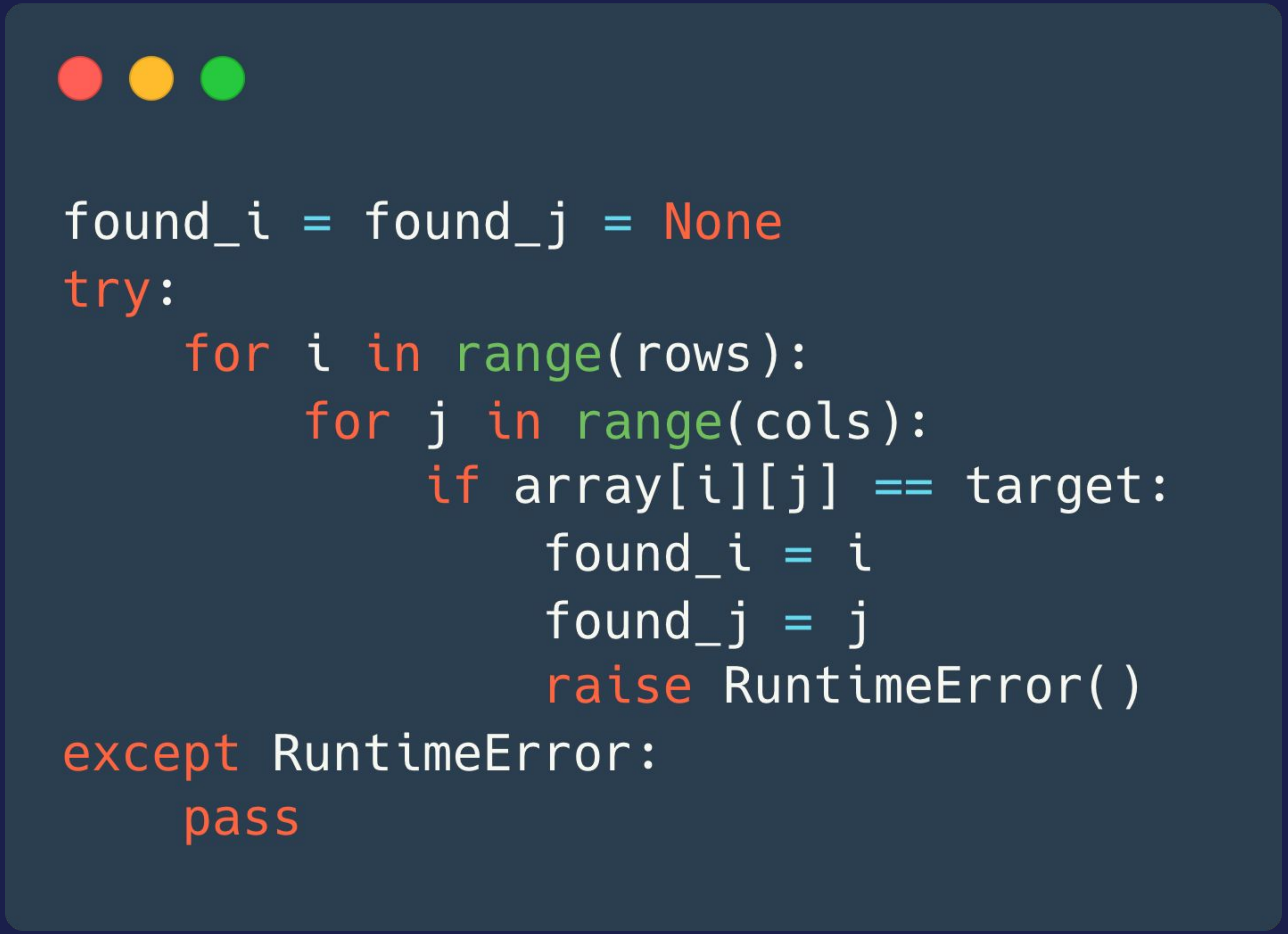


```python
for i in range(rows):
    for j in range(cols):
        if array[i][j] == target:
            found_i = i
            found_j = j
            break
    else:
        continue
    break
else:
    found_i = found_j = None
```

https://stackoverflow.com/questions/653509/breaking-out-of-nested-loops

## How about Python?

With exceptions 😈



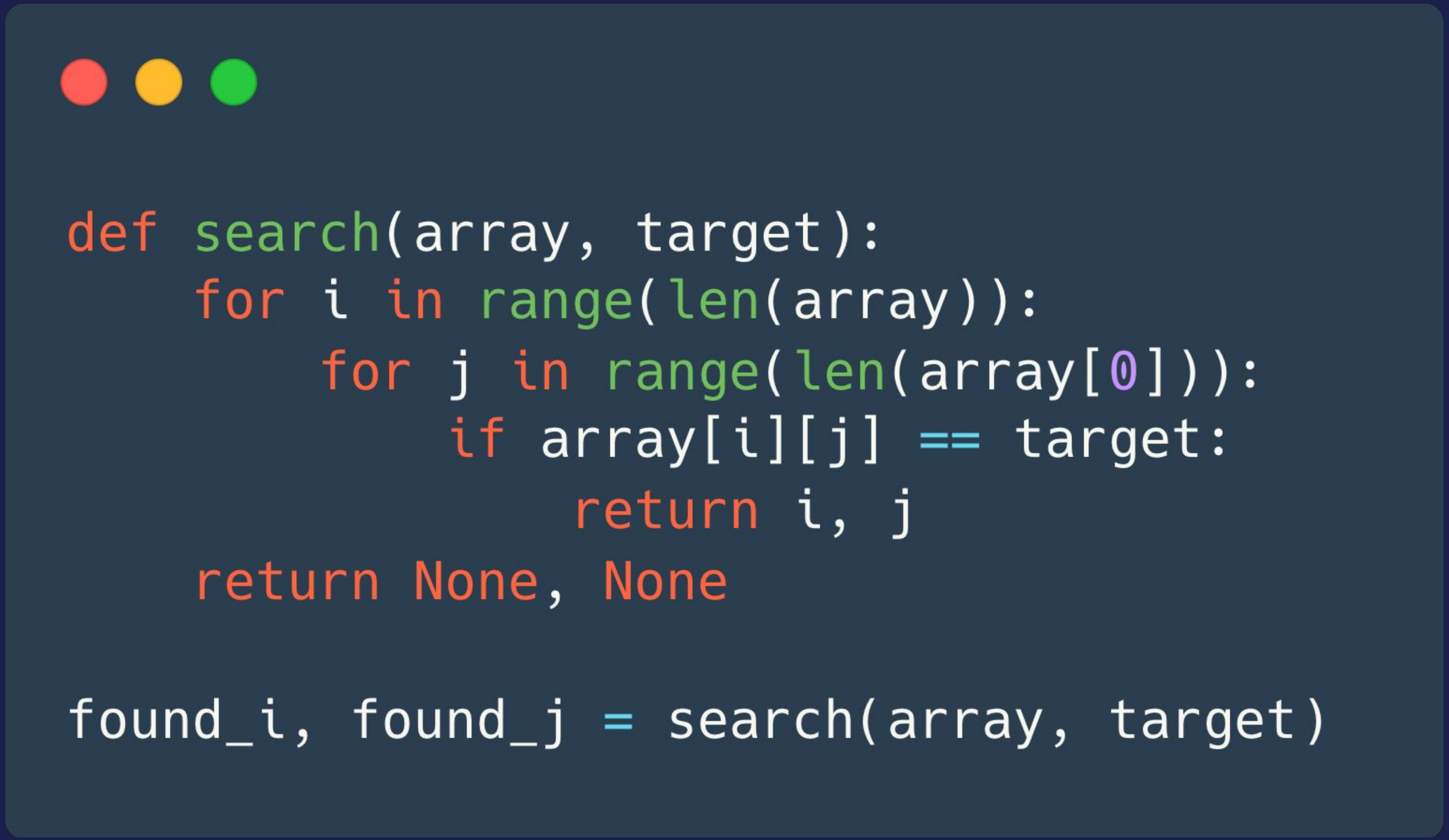
```
found_i = found_j = None
try:
    for i in range(rows):
        for j in range(cols):
            if array[i][j] == target:
                found_i = i
                found_j = j
                raise RuntimeError()
except RuntimeError:
    pass
```

https://stackoverflow.com/questions/653509/breaking-out-of-nested-loops

inovex

# How about Python?

Use functions! 🤩



```python
def search(array, target):
    for i in range(len(array)):
        for j in range(len(array[0])):
            if array[i][j] == target:
                return i, j
    return None, None

found_i, found_j = search(array, target)
```

https://stackoverflow.com/questions/653509/breaking-out-of-nested-loops

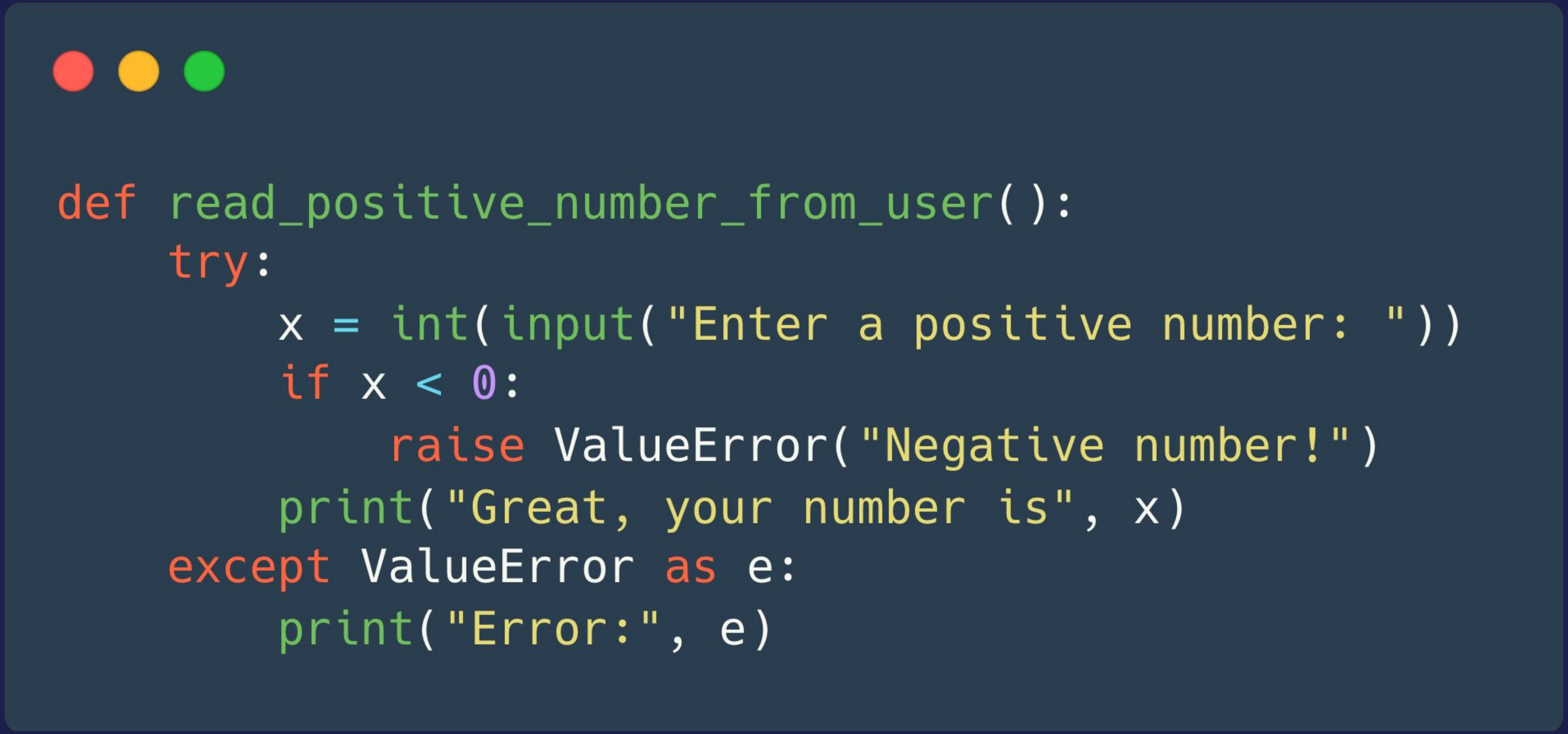
# Why Exceptions Exist and What They Are

inovex

Why Exceptions Exist and What They Are

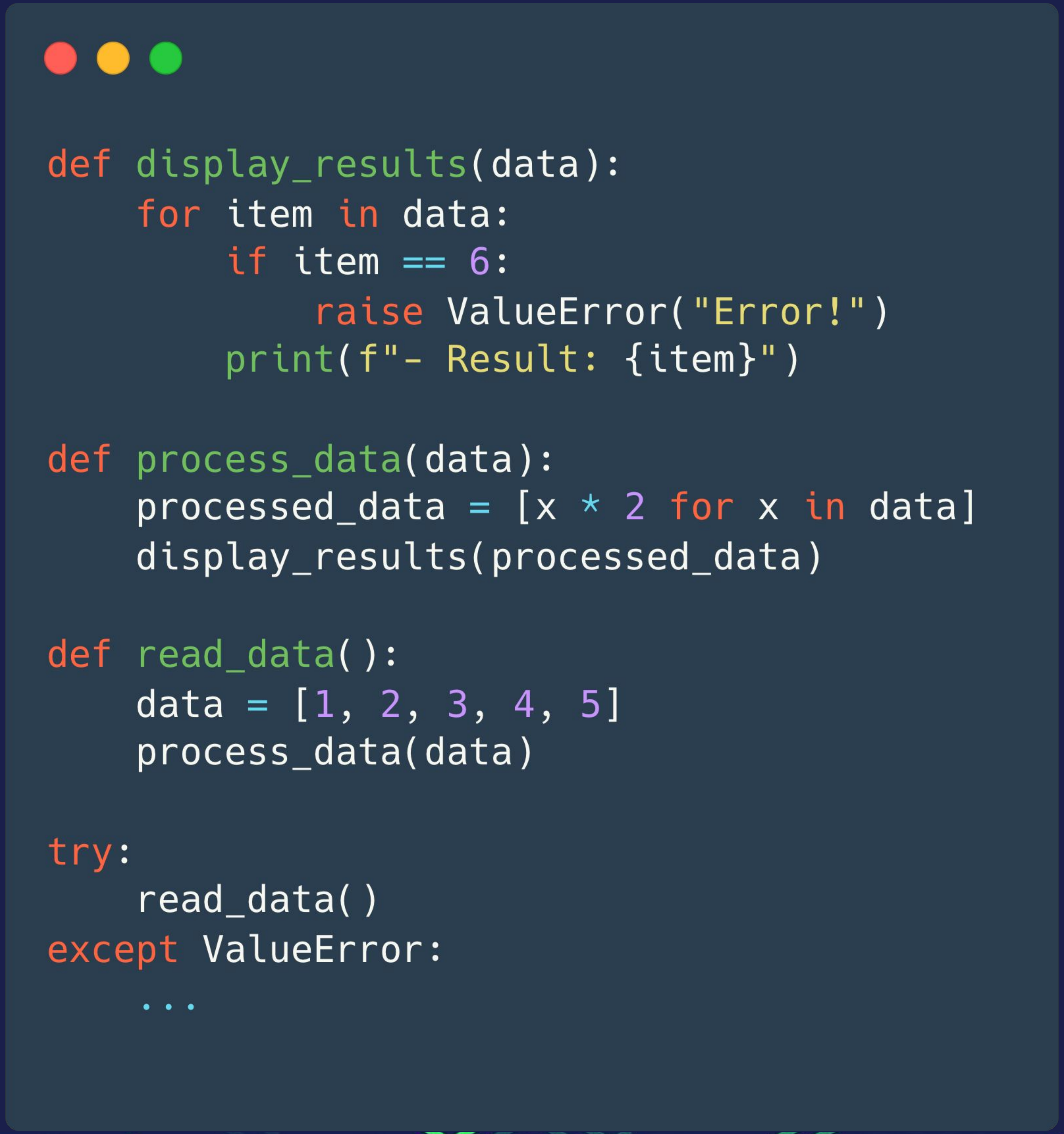
# Exceptions in Python

An exception is an event that breaks normal program flow, typically representing an error or special case requiring explicit handling.



```python
def read_positive_number_from_user():
    try:
        x = int(input("Enter a positive number: "))
        if x < 0:
            raise ValueError("Negative number!")
        print("Great, your number is", x)
    except ValueError as e:
        print("Error:", e)
```

# Climbing the Stack

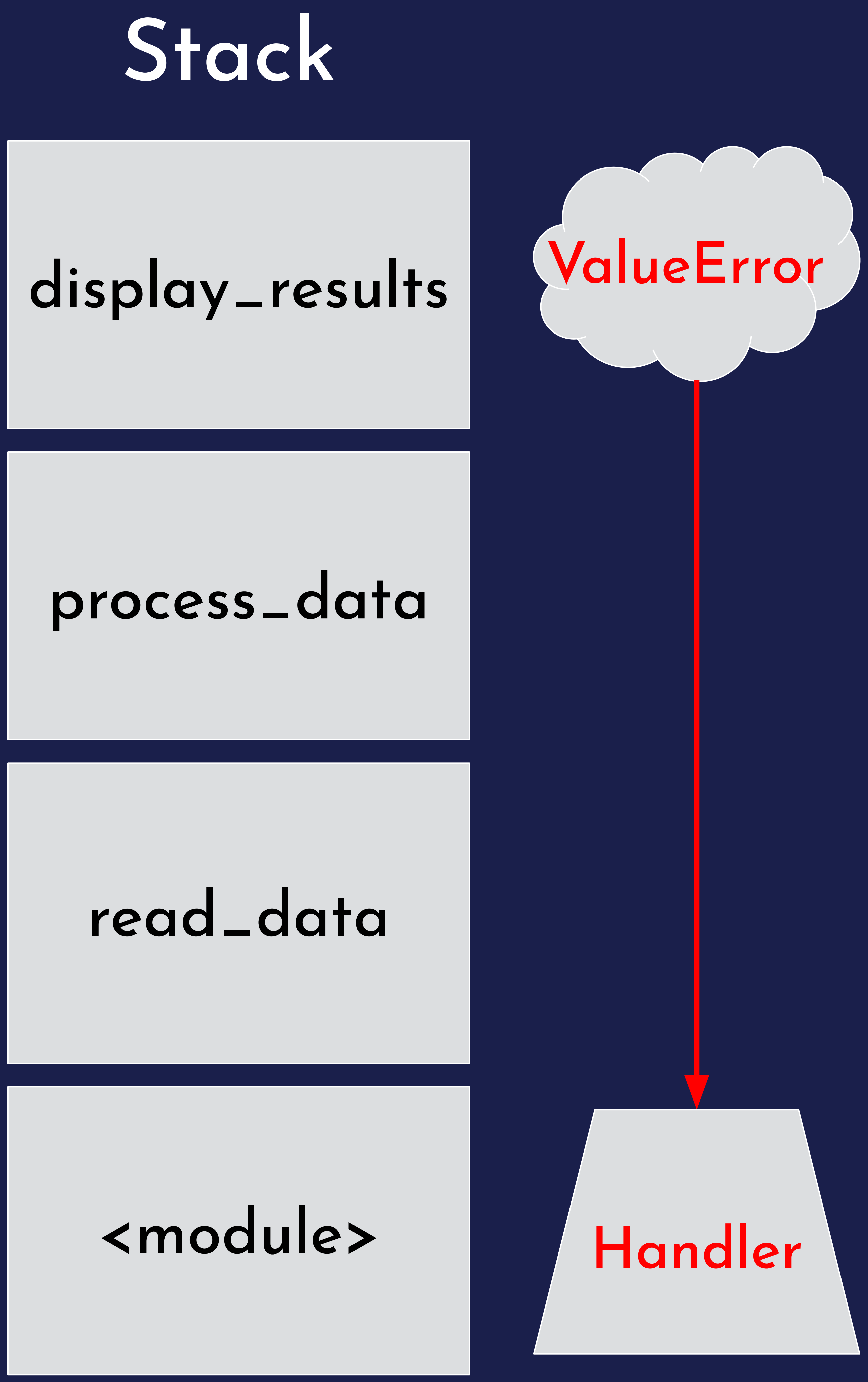


```
def display_results(data):
    for item in data:
        if item == 6:
            raise ValueError("Error!")
        print(f"- Result: {item}")

def process_data(data):
    processed_data = [x * 2 for x in data]
    display_results(processed_data)

def read_data():
    data = [1, 2, 3, 4, 5]
    process_data(data)

try:
    read_data()
except ValueError:
    ...
```

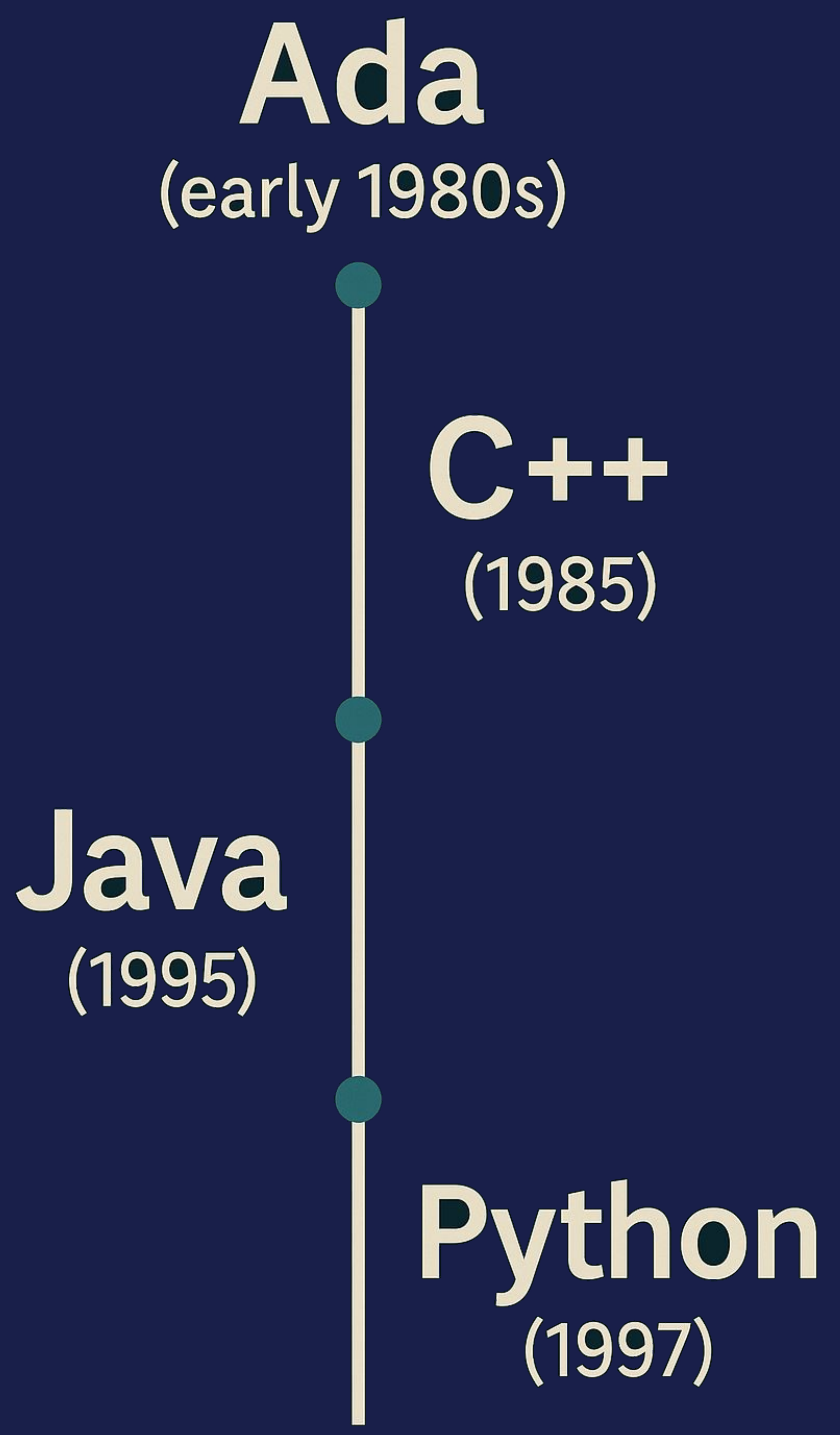


# History of Exceptions

Support of exceptions is quite common in programming languages from the 80s on.

Why?

- Separate normal logic from error handling
- Make error propagation automatic



Ada
(early 1980s)

C++
(1985)

Java
(1995)

Python
(1997)

Why Exceptions Exist and What They Are

## Problems with Exceptions

1. Invisible control flow
2. Error-handling surprises, e.g. in dependencies
3. Debugging complexity
4. Concurrency & parallelism
5. Performance & resource allocation, e.g. exceptions in C++ are discouraged.

**Goto jumps to some other line, exception goes up the stack.**

https://belaycpp.com/2021/06/16/exceptions-are-just-fancy-gotos/

inovex

# The Evolution Toward Result Types

inovex

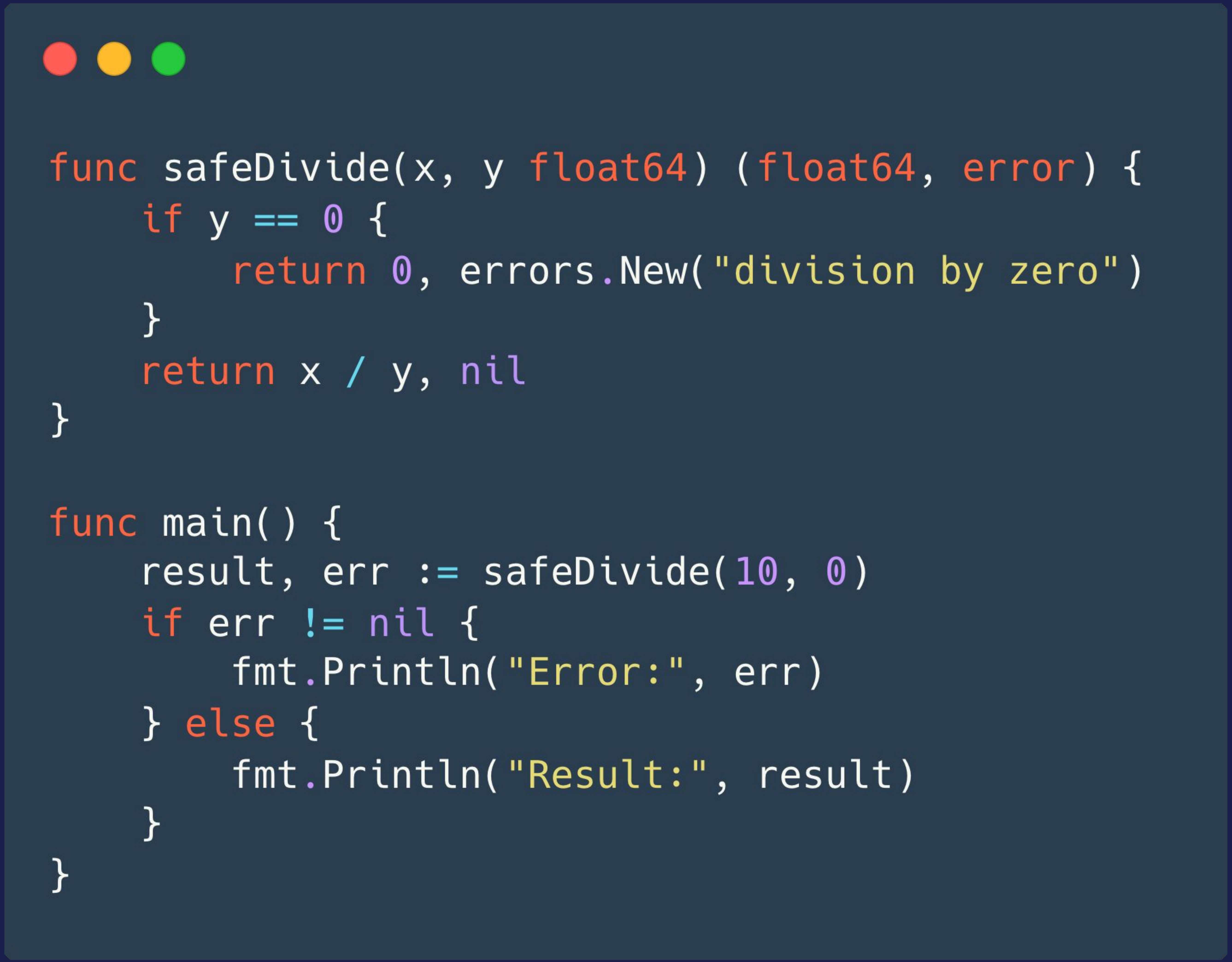
# Result Types

Return the actual value or error state wrapped in a container type and enforce handling the error state when opening the container.

- The concept of wrapping values and modeling alternatives is part of Algebraic Data Types (ADTs).
- If ADTs adhere to certain mathematical laws by implementing the monad interface, they are called **monads**.
- This concept is an important aspect in functional programming.

The Evolution Toward Result Types

# Golang (2009)



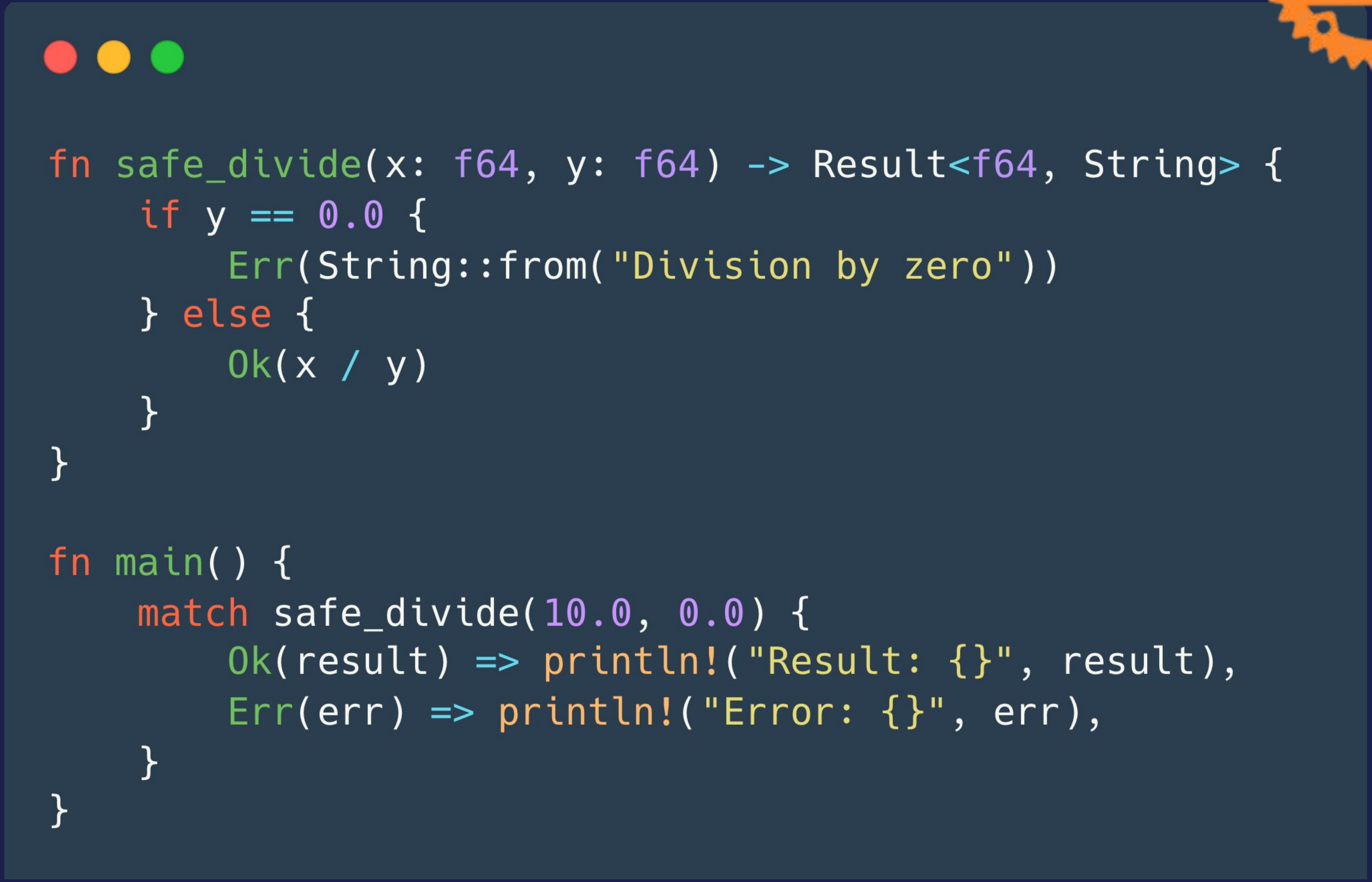
```go
func safeDivide(x, y float64) (float64, error) {
    if y == 0 {
        return 0, errors.New("division by zero")
    }
    return x / y, nil
}

func main() {
    result, err := safeDivide(10, 0)
    if err != nil {
        fmt.Println("Error:", err)
    } else {
        fmt.Println("Result:", result)
    }
}
```

inovex

The Evolution Toward Result Types

# Rust (2015)

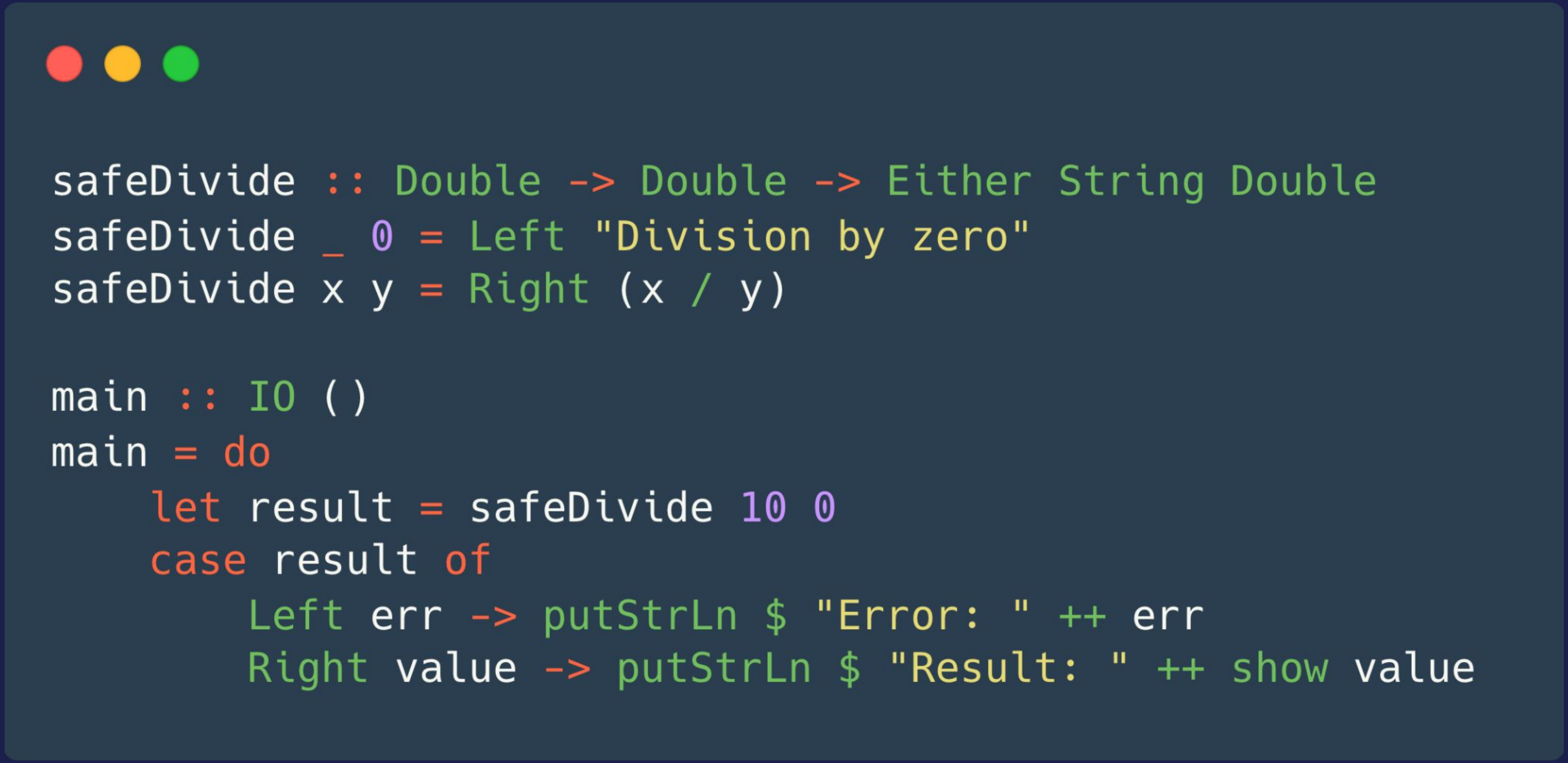


```rust
fn safe_divide(x: f64, y: f64) -> Result<f64, String> {
    if y == 0.0 {
        Err(String::from("Division by zero"))
    } else {
        Ok(x / y)
    }
}

fn main() {
    match safe_divide(10.0, 0.0) {
        Ok(result) => println!("Result: {}", result),
        Err(err) => println!("Error: {}", err),
    }
}
```

The Evolution Toward Result Types

# Haskell (1990)



```haskell
safeDivide :: Double -> Double -> Either String Double
safeDivide _ 0 = Left "Division by zero"
safeDivide x y = Right (x / y)

main :: IO ()
main = do
    let result = safeDivide 10 0
    case result of
        Left err -> putStrLn $ "Error: " ++ err
        Right value -> putStrLn $ "Result: " ++ show value
```

# What do we get from Result Types?

- No hidden control flow
- Explicitness: force the caller to handle success/failure
- Easier to reason and test code

Before After

# Using Result Types in Python

inovex

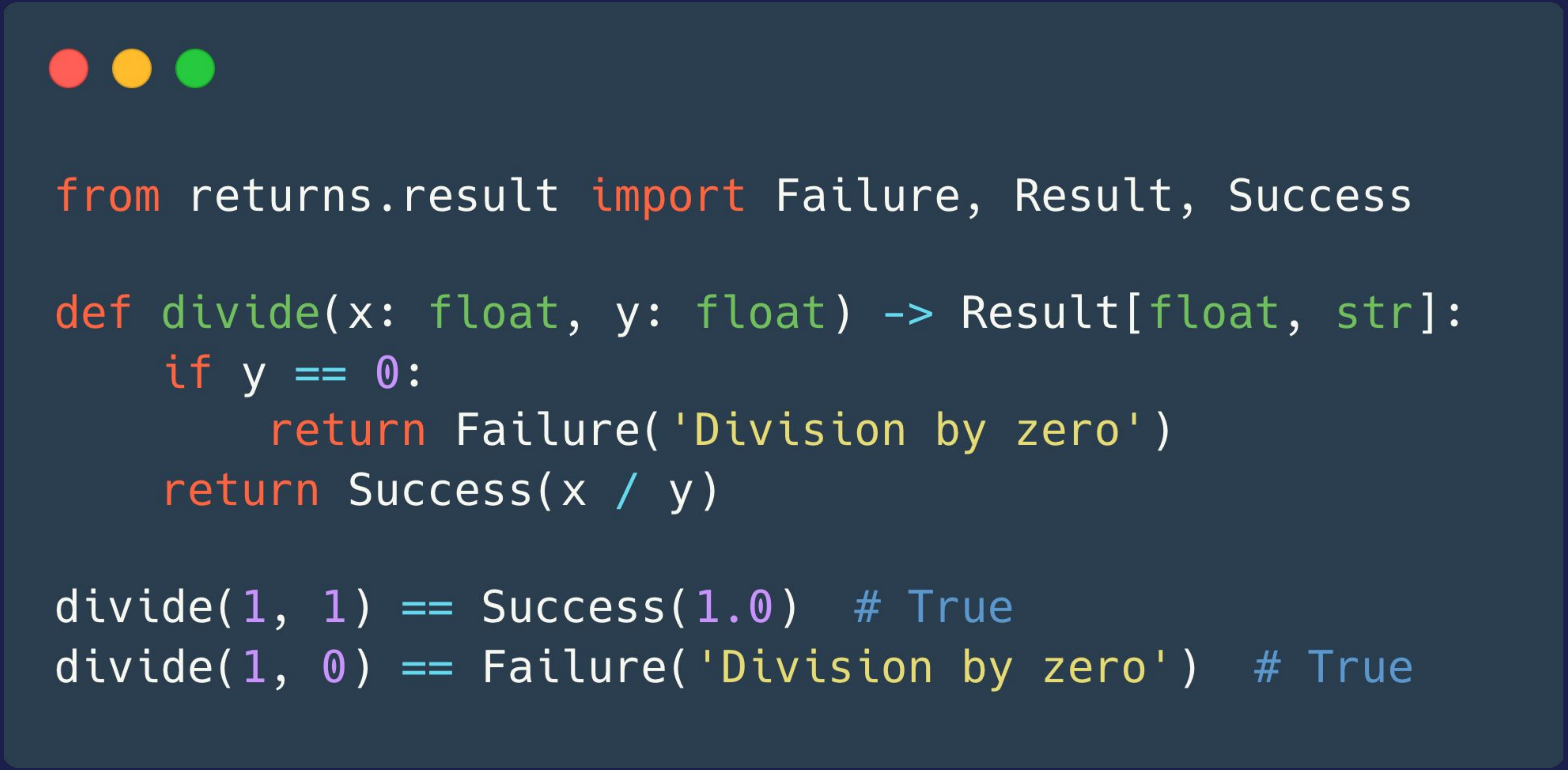
## How to use result types in Python?

Libraries offering result type containers like Maybe, Result, IO, Future, etc.

| Library | Comment | Maintained |
|---|---|---|
| returns | Haskell / FP inspired & full-featured, pythonic | ✅ |
| result | simple and rust-like | ❌ |
| oslash | Haskell-inspired | ❌ |
| expression | F# / OCaml-inspired, simplistic | ✅ |

Using Result Types in Python

## Success and Failure



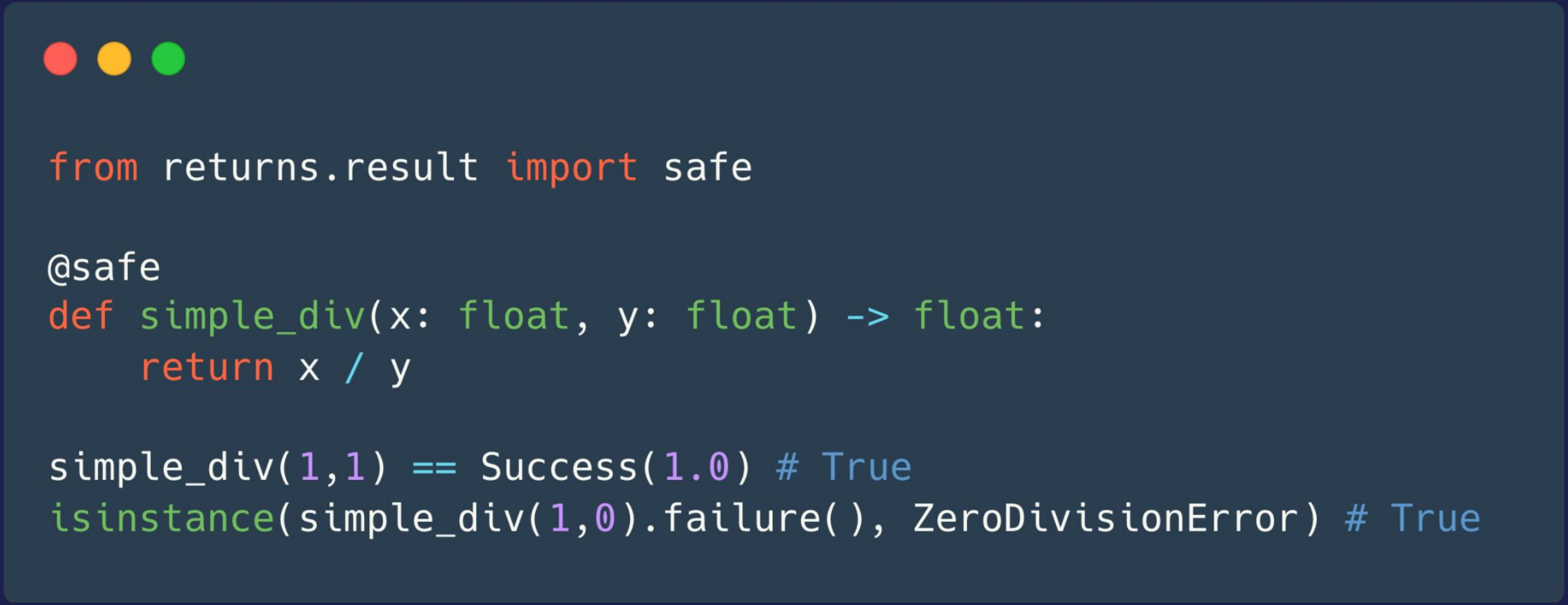
```python
from returns.result import Failure, Result, Success

def divide(x: float, y: float) -> Result[float, str]:
    if y == 0:
        return Failure('Division by zero')
    return Success(x / y)

divide(1, 1) == Success(1.0)  # True
divide(1, 0) == Failure('Division by zero')  # True
```

Using Result Types in Python

# Make functions safe by wrapping all exceptions into return types

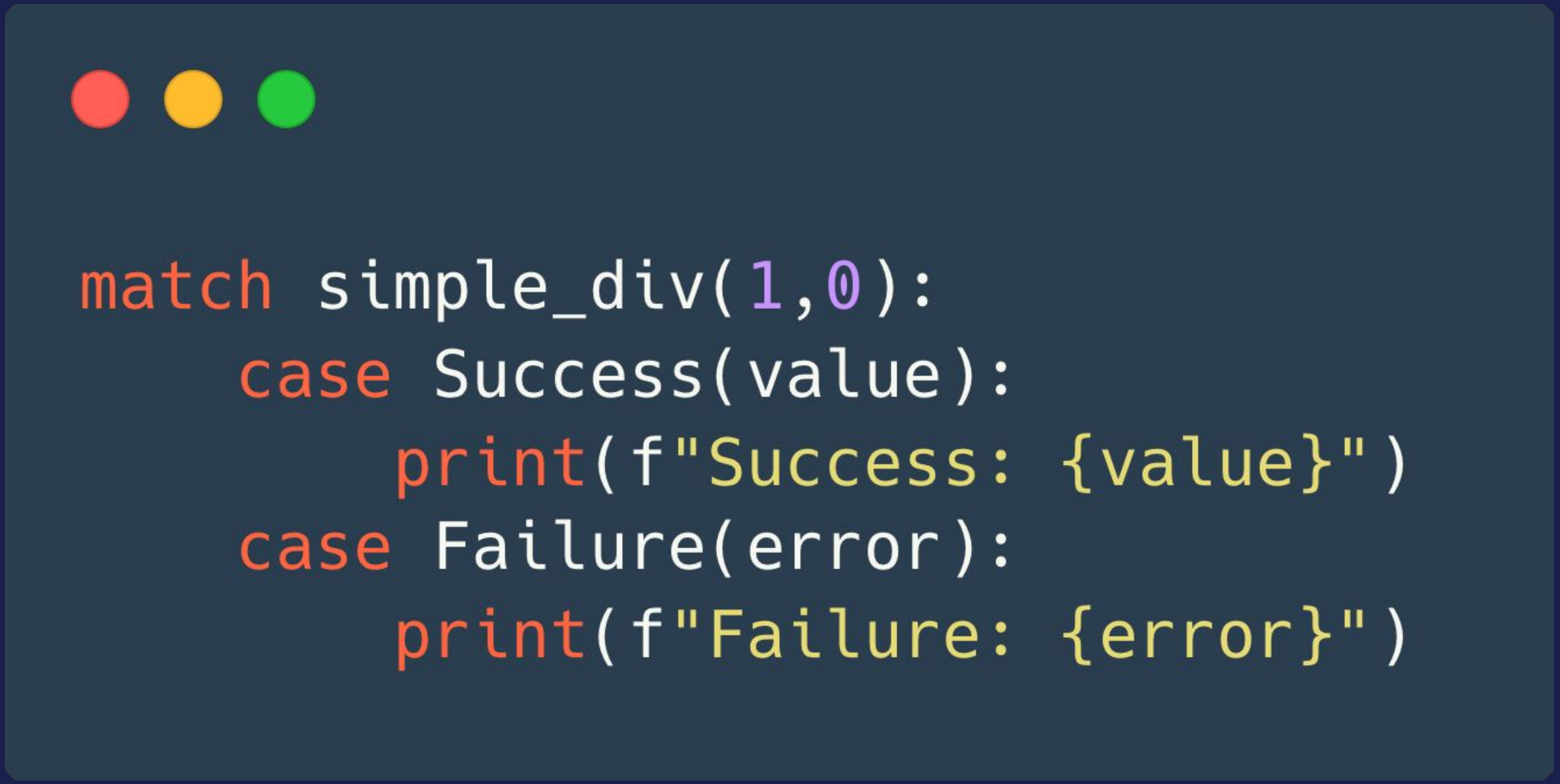


```python
from returns.result import safe

@safe
def simple_div(x: float, y: float) -> float:
    return x / y

simple_div(1,1) == Success(1.0) # True
isinstance(simple_div(1,0).failure(), ZeroDivisionError) # True
```

## Working with the wrapped values of a result type



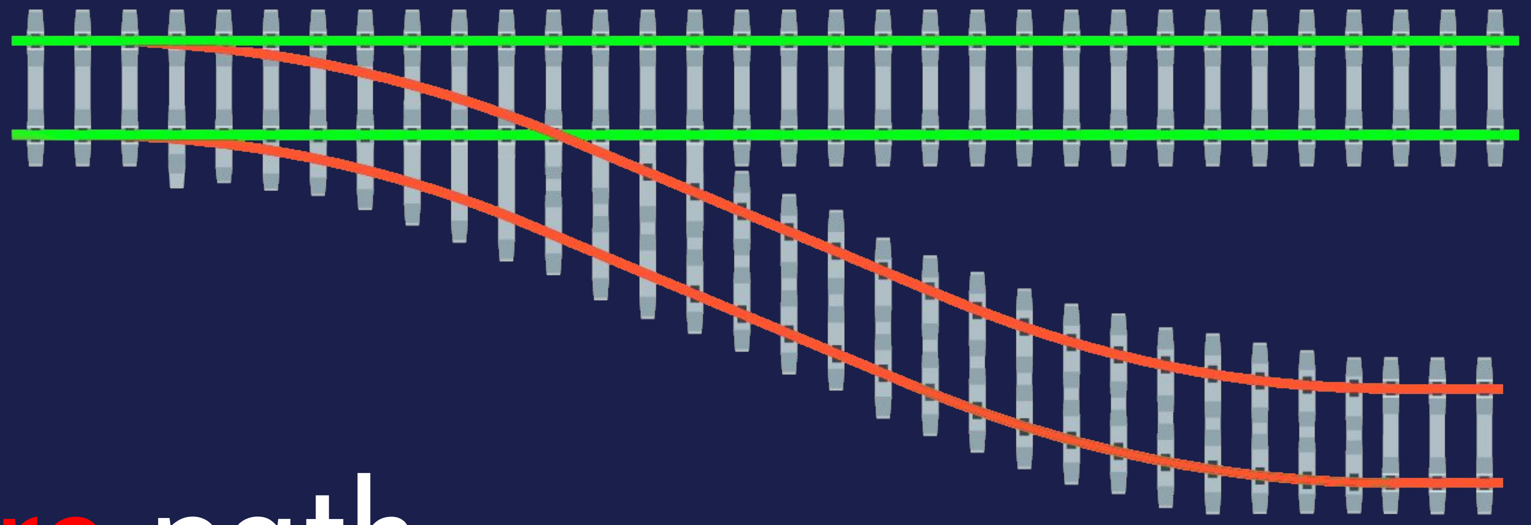
```
match simple_div(1,0):
    case Success(value):
        print(f"Success: {value}")
    case Failure(error):
        print(f"Failure: {error}")
```

Analogue to Haskell, we match Success and Failure to unwrap the value or error.

# Railway oriented programming

So we have Result Types now, how to replace exceptions now?

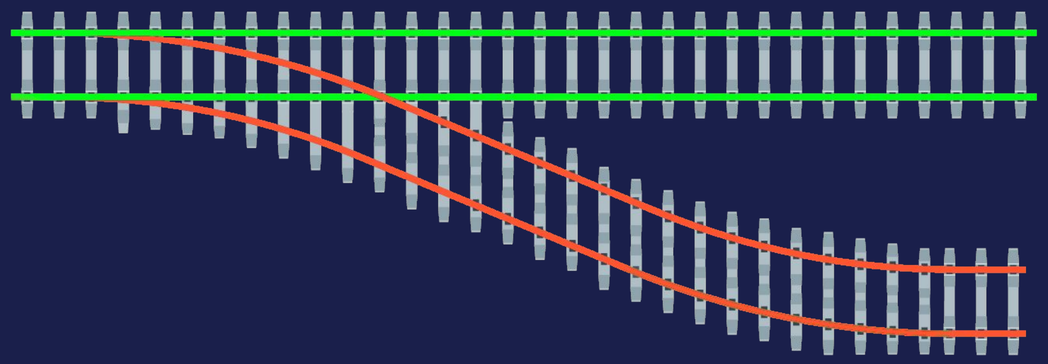
**Railway oriented Programming!**



Explicitly handling the success and failure path and by simply composing basic building blocks

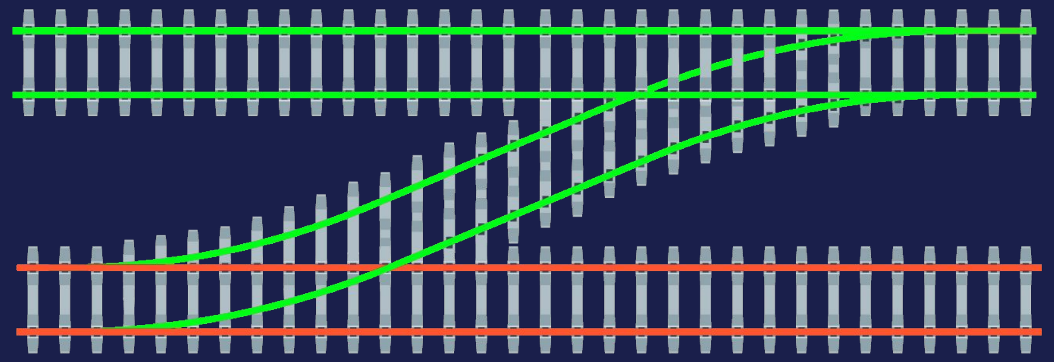
https://fsharpforfunandprofit.com/rop/ by Scott Wlaschin

# Compose basic building blocks like a five year old!



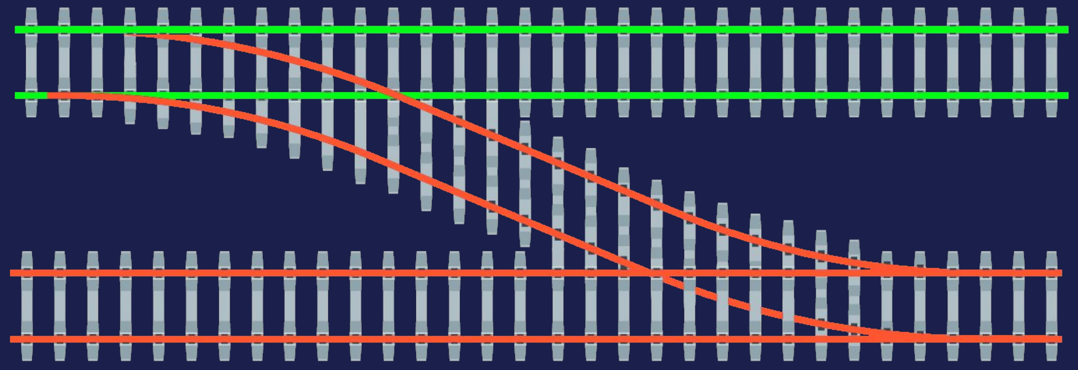
possible failure
occurs

+

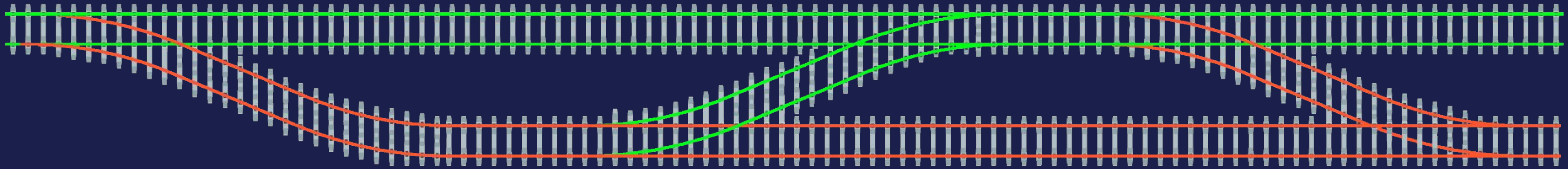


apply operation
to failure, that
may recover
from the failure

+

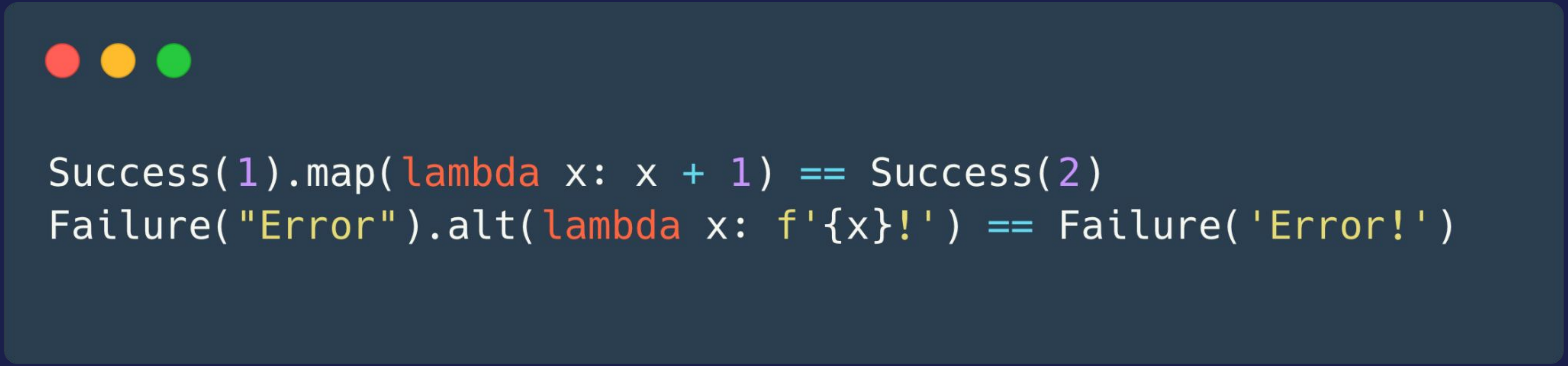


apply operation to
success value, that
may lead to an
failure

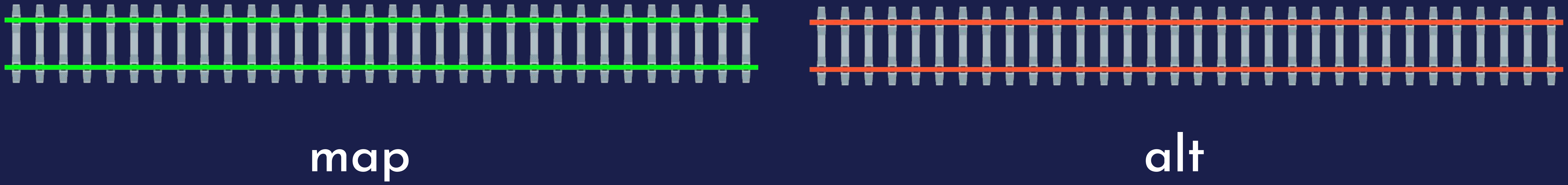


Using Result Types in Python

# map() & alt() for applying pure functions to success and failure



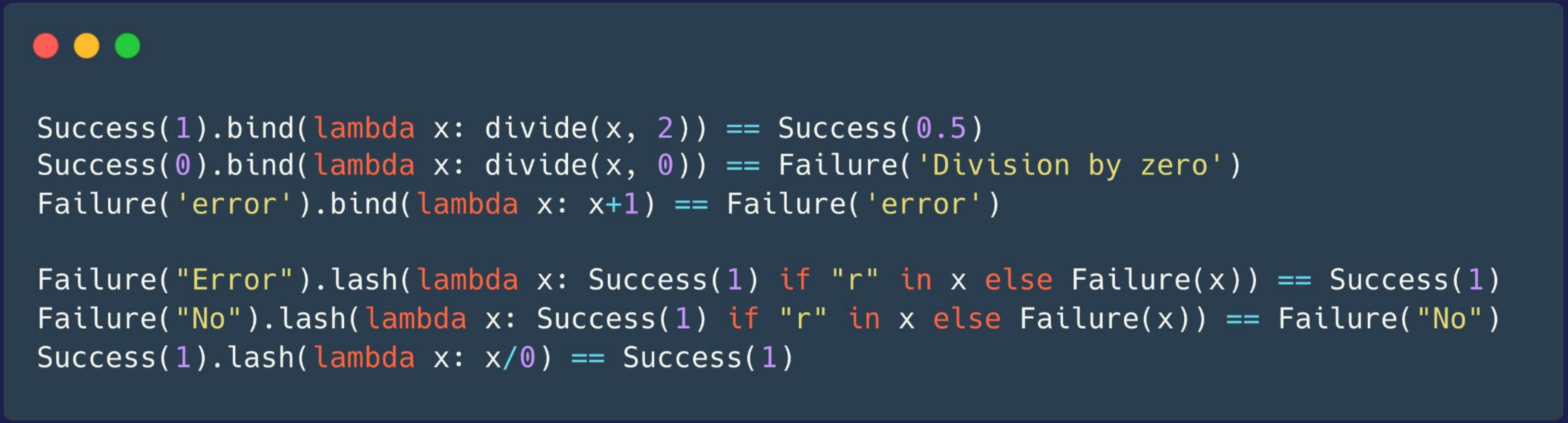
```python
Success(1).map(lambda x: x + 1) == Success(2)
Failure("Error").alt(lambda x: f'{x}!') == Failure('Error!')
```



inovex

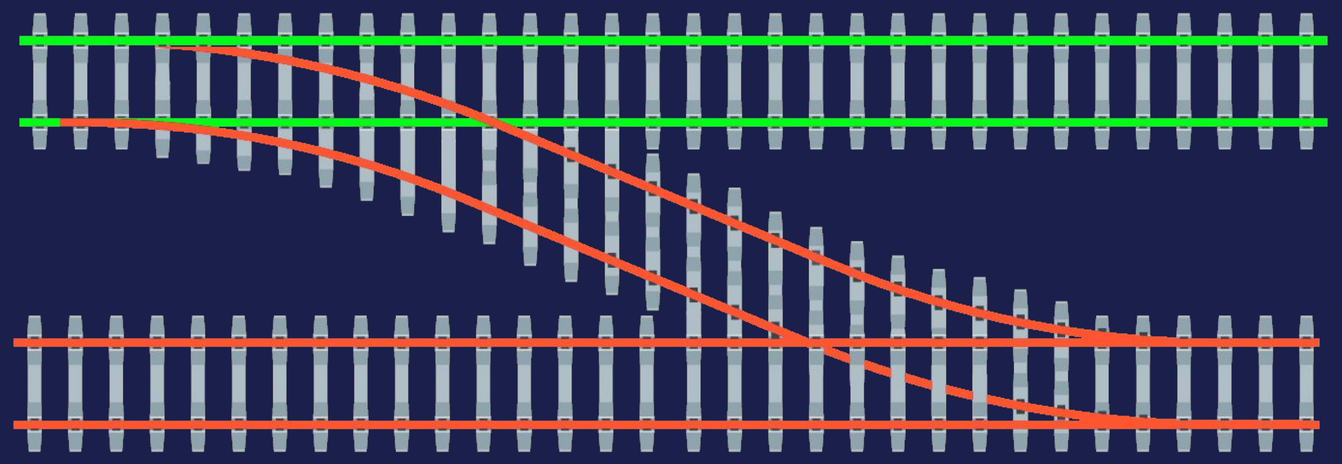
Using Result Types in Python

## bind() & lash() for applying non-pure functions to success and failure

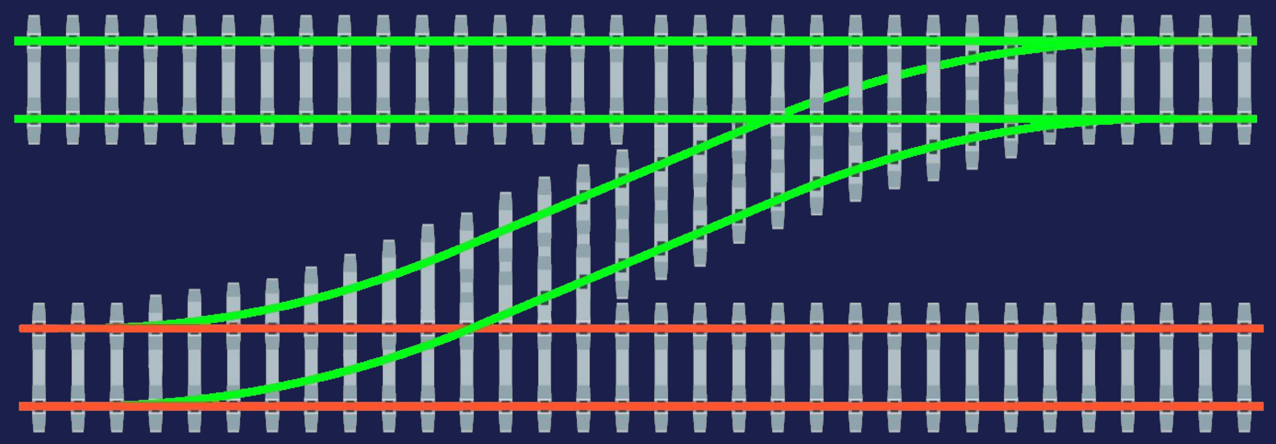


```
Success(1).bind(lambda x: divide(x, 2)) == Success(0.5)
Success(0).bind(lambda x: divide(x, 0)) == Failure('Division by zero')
Failure('error').bind(lambda x: x+1) == Failure('error')

Failure("Error").lash(lambda x: Success(1) if "r" in x else Failure(x)) == Success(1)
Failure("No").lash(lambda x: Success(1) if "r" in x else Failure(x)) == Failure("No")
Success(1).lash(lambda x: x/0) == Success(1)
```



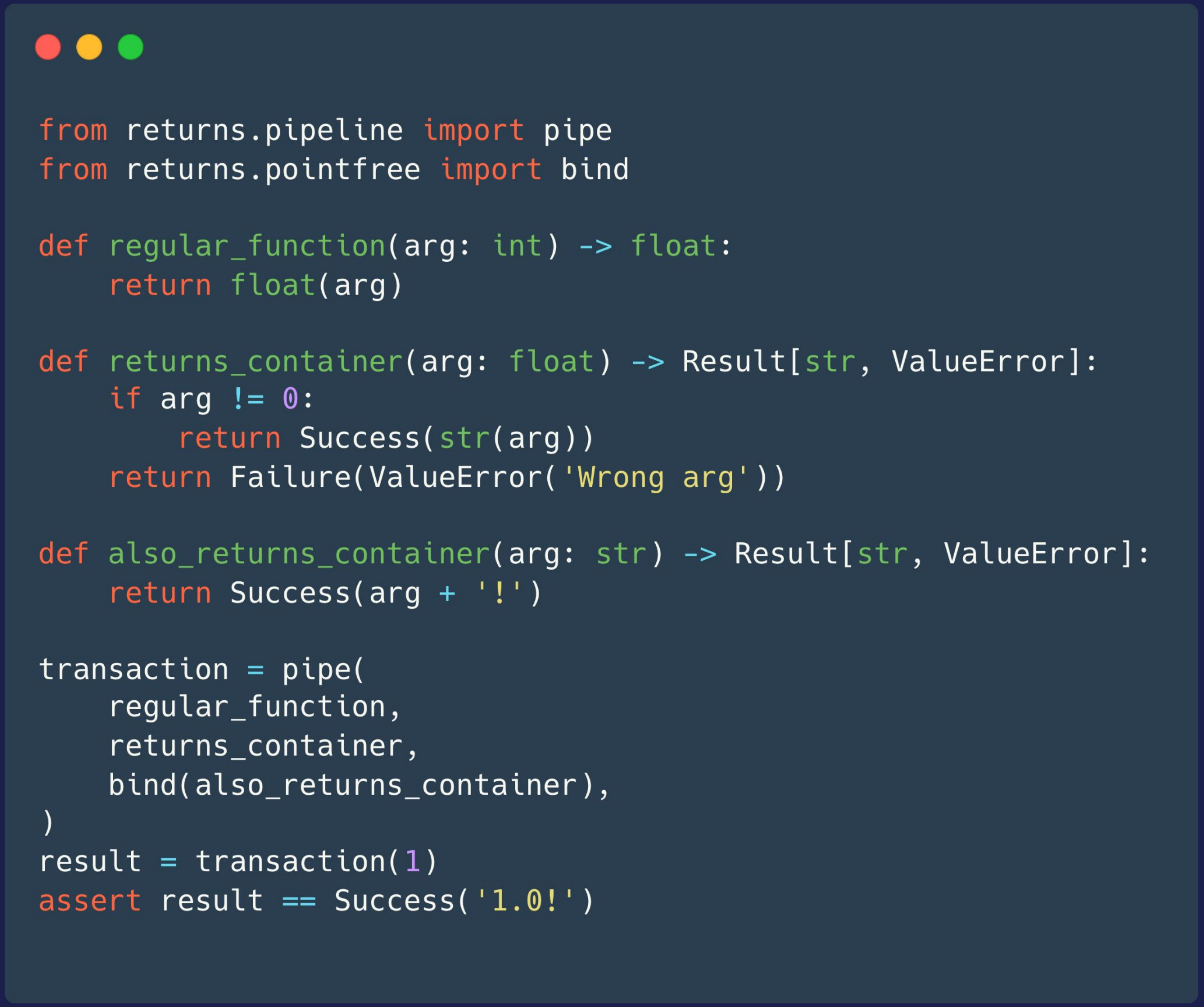
bind



lash

inovex

# Composition with pipe(...)



```python
from returns.pipeline import pipe
from returns.pointfree import bind

def regular_function(arg: int) -> float:
    return float(arg)

def returns_container(arg: float) -> Result[str, ValueError]:
    if arg != 0:
        return Success(str(arg))
    return Failure(ValueError('Wrong arg'))

def also_returns_container(arg: str) -> Result[str, ValueError]:
    return Success(arg + '!')

transaction = pipe(
    regular_function,
    returns_container,
    bind(also_returns_container),
)
result = transaction(1)
assert result == Success('1.0!')
```

## What else can be done with returns?

- **Containers** for IO, Futures (async calls), etc.
- **Managed** for dealing with resources (functional counterpart of context manager)
- Many more **compositions** besides pipe to deal with result types
- Dealing with variadic, i.e. non-unary, functions with helpers like (un-)curry, partial, do-notation, etc.
- **Trampolines** for Tail Call Optimization
- and more....

## Conclusion

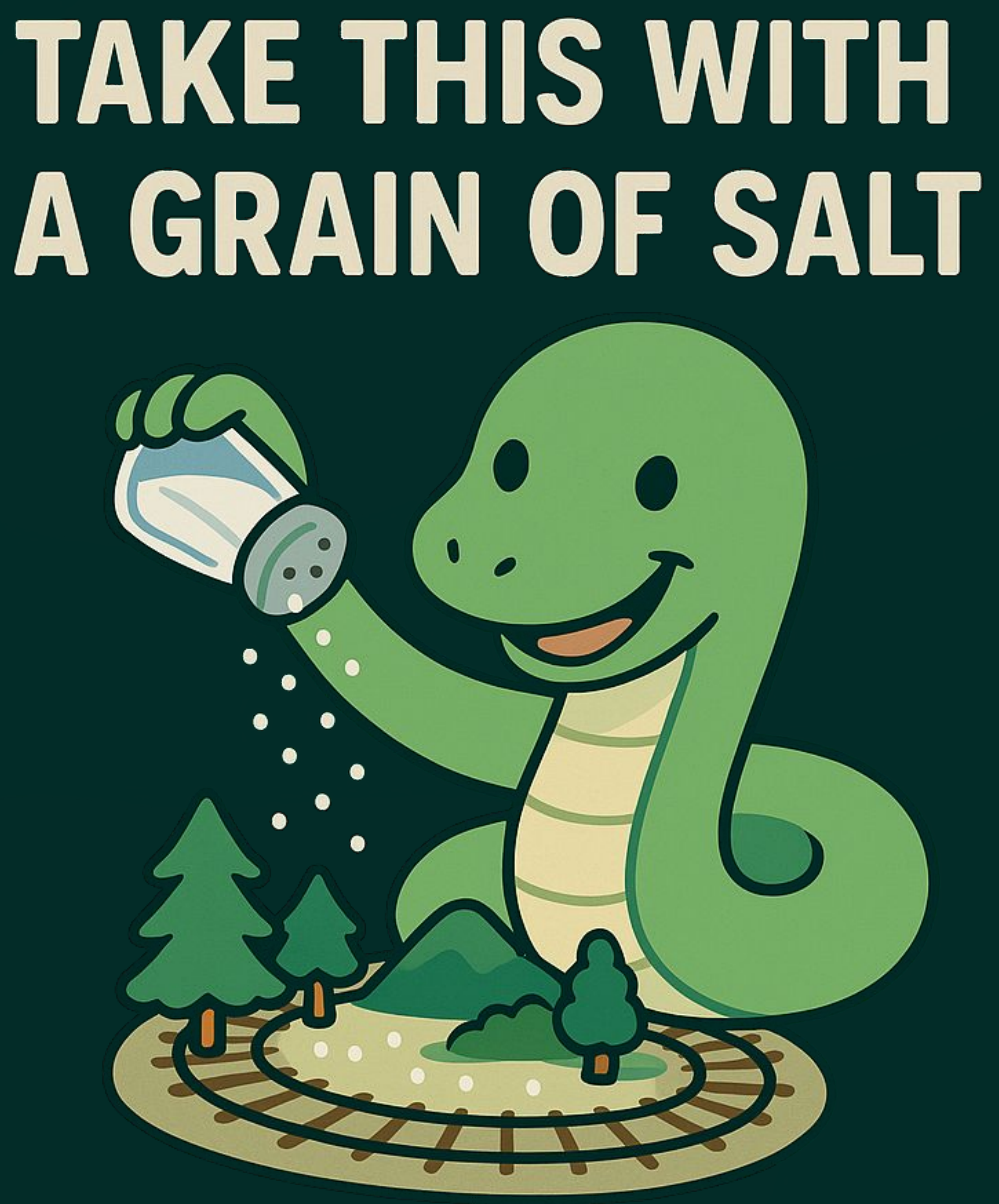
- Also consider the failure path! Not just the happy path of your program.
- How Algebraic Data Types, like Result, work conceptually
- Railway-oriented programming as a concept that replaces traditional exception handling.
- Advanced (4th-generation) languages like Rust & Haskell enforce the usage of result types

inovex

Conclusion

## So should you apply this now in your next Python project?

- Python is not inherently functional, and over-applying functional paradigms can make code less readable and idiomatic.
- returns might be the right tool for certain use-cases if your team is and thinks functional



PyConDE & PyData 2025

# Thank you!

Dr. Florian Wilhelm

Head of Data Science & Mathematical Modelling

florian.wilhelm@inovex.de

inovex.de

@inovexlife

@inovexgmbh

inovex