Beyond Try-Except: Python's Frontier of Error Handling with Monads and Railway Magic

by Sebas Arias

Hi! I'm Sebastian 👨‍💻 🇪🇨

Software Engineer at **Stack Builders**

LBYL and **EAFP** Overview

Look Before You Leap

```
if 'key' in my_dict:
    value = my_dict['key']
else:
    # Handle the missing key
```

Strengths

Predictable

Handle Conditions

Prevention

Avoid exceptions

Weakness

Verbose

Result in more code

Race Conditions

Changes between checks and operations

Easier to Ask for Forgiveness than Permission

This is a more "Pythonic" and optimistic approach

```
try:
    value = my_dict['key']
except KeyError:
    # Handle the missing key
```

Strengths

Concise

Less code

Concurrency Friendly

Avoid race conditions

Weakness

Exception Overhead

Raise/Catch has slower performance

Less Explicit

Not clear what edge cases are being considered

Why **This Matters** in Python?

Simple is Better than Complex

Readability Counts

Zen of Python

Challenges with the Traditional Approach

As the **Codebase Grow in Complexity**, We Encounter...

1 **Scalability**

Each check adds to the code's complexity, making it harder to read and increasing the chances of missing edge cases, and relying on `try/except` blocks can make your code harder to understand.

2 **Error Propagation and Handling**

Errors often need to be propagated up through several layers of functions or modules. Catching them at the right level without losing context becomes challenging.

3 **Managing Null Values and Edge Cases**

Both LBYL and EAFP can struggle with scenarios like null values or unexpected input.

4 **Readability and Maintainability**

Maintaining clear and readable error-handling code becomes crucial. Both LBYL and EAFP make the core logic harder to follow and to understand the intended behaviour

The **Need** For More **Robust Error Handling** Strategy

Consistency

How to handle errors across the codebase

Clarity

Make it clear where and why an error occurs

Null Safety

Prevent null values and similar issues from slipping through without proper handling

Introducing Monads

> A monad is a design pattern used to encapsulate computations represented as a series of steps.

Learn You a Haskell for Great Good! by Miran Lipovača

In other words, a monad is...

- Design Pattern
- Encapsulates Values and Computations
- Consistent and Predictable Way

How Monad(ic) Error Handling Work

1

Encapsulate the Operation Result

It can be a **Success or** an **Error**

2

Chaining Operations

Allows **chaining operations** together **in a clean and predictable way**

3

Results

The outcome becomes predictable. Either a success or an Error

Paradigm Shift. Introducing **Result Monad**

> The **Result** monad **represents** either **success (Ok)** or **failure (Err)**. It is a way of encoding the idea that a computation might produce a valid result or it might fail with an error. (…) the Result monad allows chaining operations that might fail, propagating errors through the computation.

Programming Rust: Fast, Safe Systems Development by Jim Blandy and Jason Orendorff.

The Result Library

Result library provides a simple and effective implementation of Result Monad.

It wraps the outcome of an operation in a `Result` object, which can be either:

Ok(value)

Representing a successful operation

Err(error)

Represents a failure

```
from result import Ok, Err

def divide(a, b):
    if b == 0:
        return Err("Division by zero")
    else:
        return Ok(a / b)

result = divide(10, 0)
```

```
result = divide(10, 2).map(lambda x: x * 2).unwrap_or("Error occurred")
```

Benefits of using **Result** Monad Library

Clarity and Readability

Composability

Null and Edge Case Problems

Consistency

Railway-Driven Development 🚂

> ***R**ailway-**O**riented **P**rogramming metaphor in Functional Programming*

RDD Key concepts

Two Tracks

Success Track

The "train" moves along the track were everything is working 😊

Failure Track

When something fails, the "train" is diverted to this track handles the error

Switching Tracks

Check the track to follow

At each step, it checks whether to stay. On success track or failure track.

This is were monadic error handling comes into play.

Functional Programming in Action!

Final Takeaway

By adopting monadic error handling and Railway-Driven Development you will

- Create cleaner, more maintainable code
- Code that scales well as the project grow in complexity
- Cognitive load is reduced by simplifying error management leading to more robust and reliable software

Questions 🙋

Let's Continue the Conversation!

linkedin.com/in/sebasarias/

Larox/monadicErrorHandlingPython

Made with Gamma