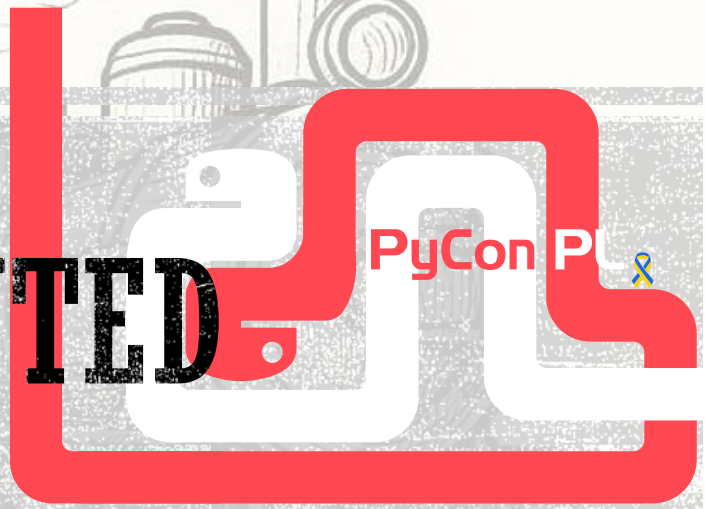
RAILWAY ORIENTED PROGRAMMING



Nowoczesna obsługa błędów w języku Python przy użyciu... torów :)

Artur Lew,
PyCon PL 2025

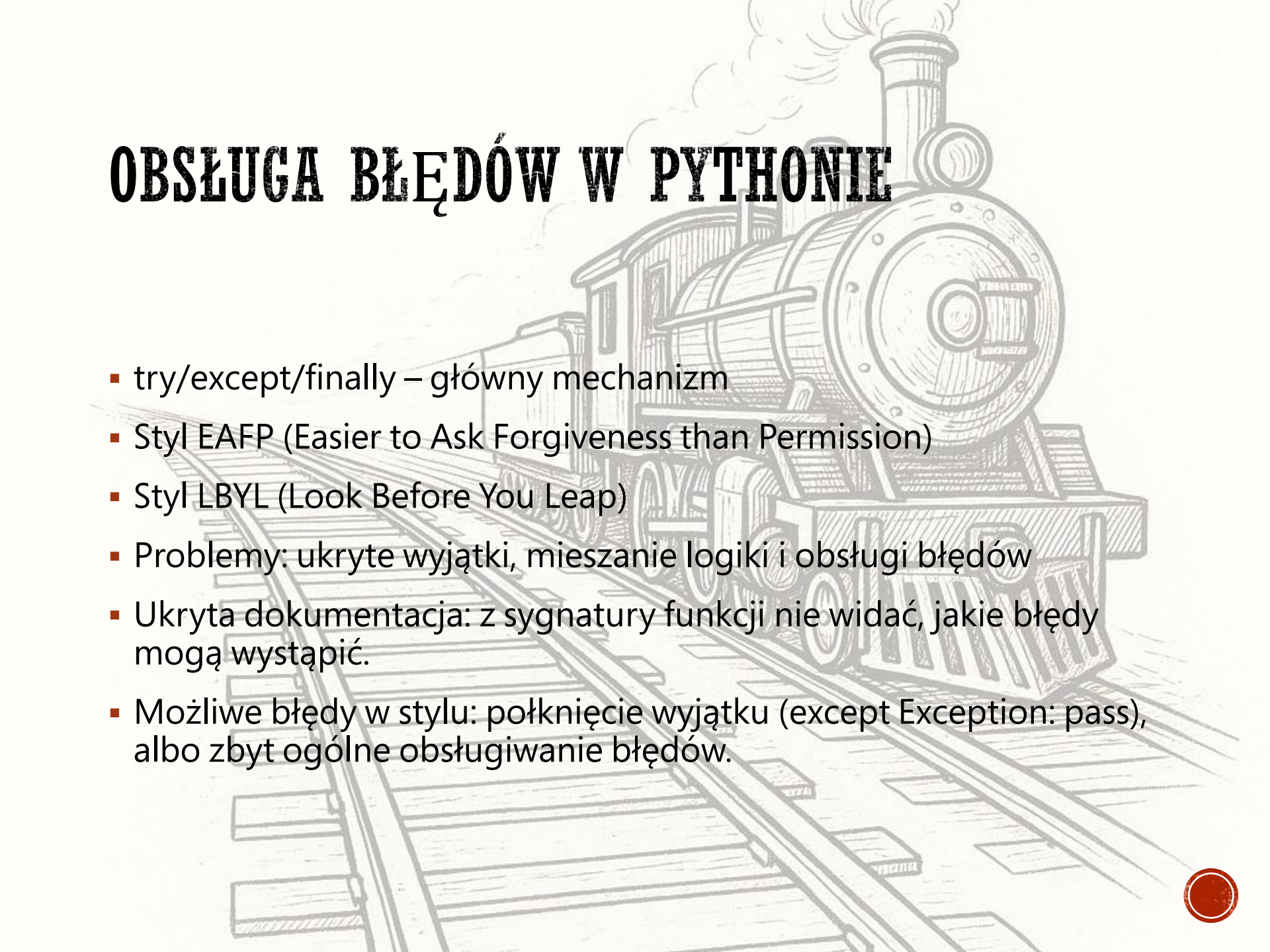
Gliwice 30.08

TEMATYKA NA DZISIAJ

- 1. Obsługa błędów w Pythonie i innych językach
- 2. Wprowadzenie do wzorca ROP
- 3. Biblioteka returns
- 4. ROP w aplikacjach synchronicznych
- 5. ROP w aplikacjach asynchronicznych
- 6. Podsumowanie

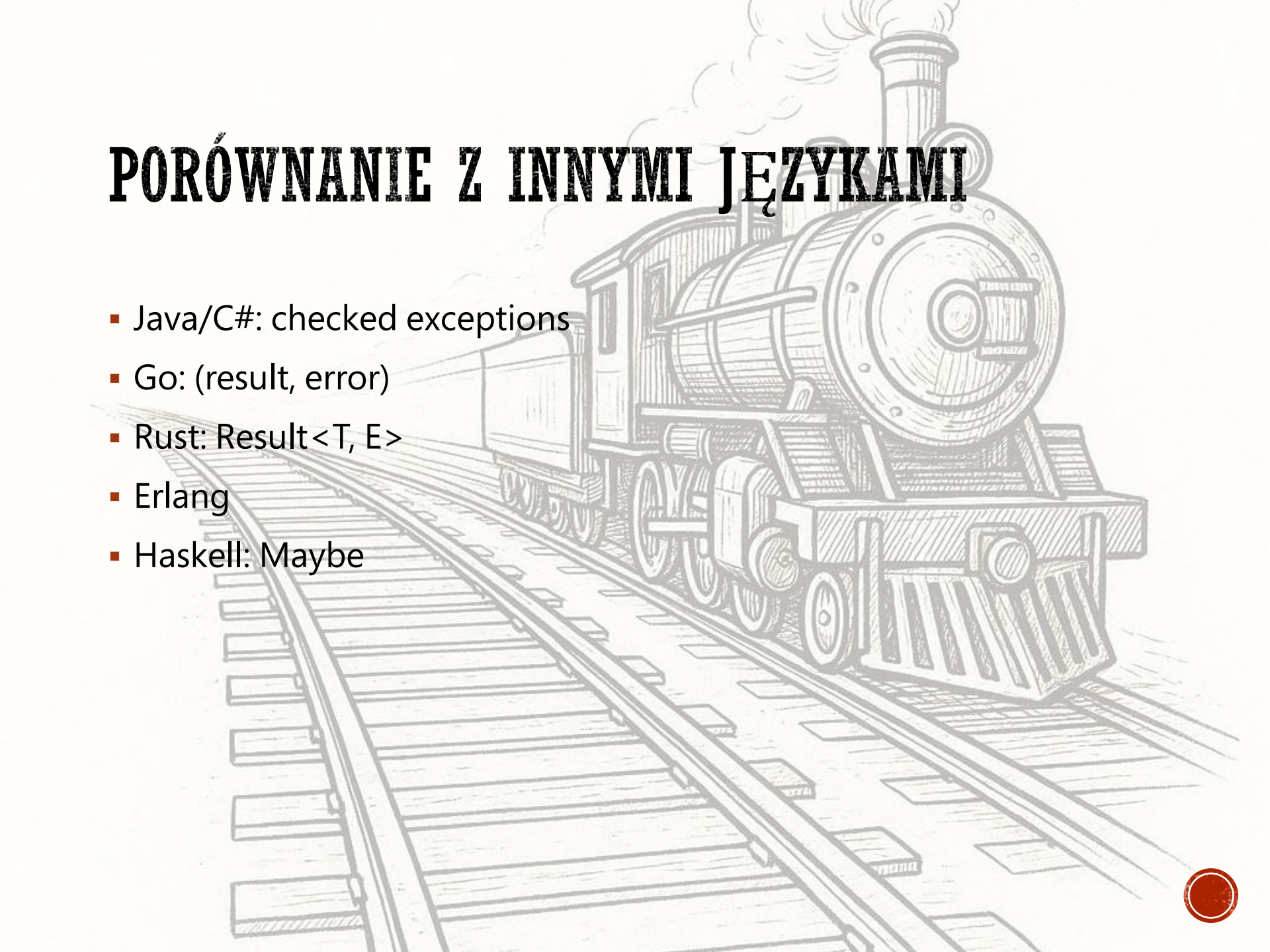
Obsługa błędów w Pythonie

- try/except/finally – główny mechanizm
- Styl EAFP (Easier to Ask Forgiveness than Permission)
- Styl LBYL (Look Before You Leap)
- Problemy: ukryte wyjątki, mieszanie logiki i obsługi błędów
- Ukryta dokumentacja: z sygnatury funkcji nie widać, jakie błędy mogą wystąpić.
- Możliwe błędy w stylu: połknięcie wyjątku (except Exception: pass), albo zbyt ogólne obsługiwanie błędów.



Porównanie z innymi językami

- Java/C#: checked exceptions
- Go: (result, error)
- Rust: Result<T, E>
- Erlang
- Haskell: Maybe



Porównanie z innymi językami

- Java/C#

```
try {
    int result = divide(10, 0);
} catch (ArithmeticException e) {
    System.out.println("Error: " + e.getMessage());
}
```

Porównanie z innymi językami

Class Exception

java.lang.Object
    java.lang.Throwable
        java.lang.Exception

Źródło:
https://download.java.net/java/early_access/valhalla/docs/api/java.base/java/lang/Exception.html

All Implemented Interfaces:

Serializable

Direct Known Subclasses:

AbsentInformationException, AgentInitializationException, AgentLoadException, AlreadyBoundException, AttachNotSupportedException, AWTException, BackingStoreException, BadAttributeValueExpException, BadBinaryOpValueExpException, BadLocationException, BadStringOperationException, BrokenBarrierException, CardException, CertificateException, ClassNotLoadedException, CloneNotSupportedException, DataFormatException, DatatypeConfigurationException, DestroyFailedException, ExecutionControl.ExecutionControlException, ExecutionException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalClassFormatException, IllegalConnectorArgumentsException, IncompatibleThreadStateException, InterruptedException, IntrospectionException, InvalidApplicationException, InvalidMidiDataException, InvalidPreferencesFormatException, InvalidTargetObjectTypeException, InvalidTypeException, InvocationException, IOException, JMException, JShellException, KeySelectorException, LambdaConversionException, LineUnavailableException, MarshalException, MidiUnavailableException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NotBoundException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, ReflectiveOperationException, RefreshFailedException, RuntimeException, SAXException, ScriptException, ServerNotActiveException, SQLException, StringConcatException, TimeoutException, TooManyListenersException, TransformerException, TransformException, UnmodifiableClassException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URIReferenceException, URISyntaxException, VMStartException, XAException, XMLParseException, XMLSignatureException, XMLStreamException, XPathException

Porównanie z innymi językami

- Scala

```scala
import scala.util.{Try, Success, Failure}

def divide(a: Int, b: Int): Try[Int] = Try(a / b)

divide(10, 0) match {
  case Success(res) => println(s"Result: $res")
  case Failure(err) => println(s"Error: ${err.getMessage}")
}
```

Porównanie z innymi językami

Go ➡️ Brak wyjątków, zwracanie (value, error) – jawne i proste.

```
func divide(a, b int) (int, error) {
    if b == 0 {
        return 0, fmt.Errorf("division by zero")
    }
    return a / b, nil
}

result, err := divide(10, 0)
if err != nil {
    fmt.Println("Error:", err)
}
```

Porównanie z innymi językami

- Rust ➡️ Result<T, E>, wymusza jawne obsłużenie błędów.

```rust
fn divide(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err("division by zero".to_string())
    } else {
        Ok(a / b)
    }
}

fn main() {
    match divide(10, 0) {
        Ok(res) => println!("Result: {}", res),
        Err(e) => println!("Error: {}", e),
    }
}
```

Porównanie z innymi językami

- Erlang ➡️ Krotki {ok, Val} | {error, Reason}, brak klasycznych wyjątków.

```
divide(A, 0) -> {error, division_by_zero};
divide(A, B) -> {ok, A div B}.

case divide(10, 0) of
    {ok, Result} -> io:format("Result: ~p~n", [Result]);
    {error, Reason} -> io:format("Error: ~p~n", [Reason])
end.
```

Porównanie z innymi językami

- Haskell - Either e a, czysto funkcyjne podejście.

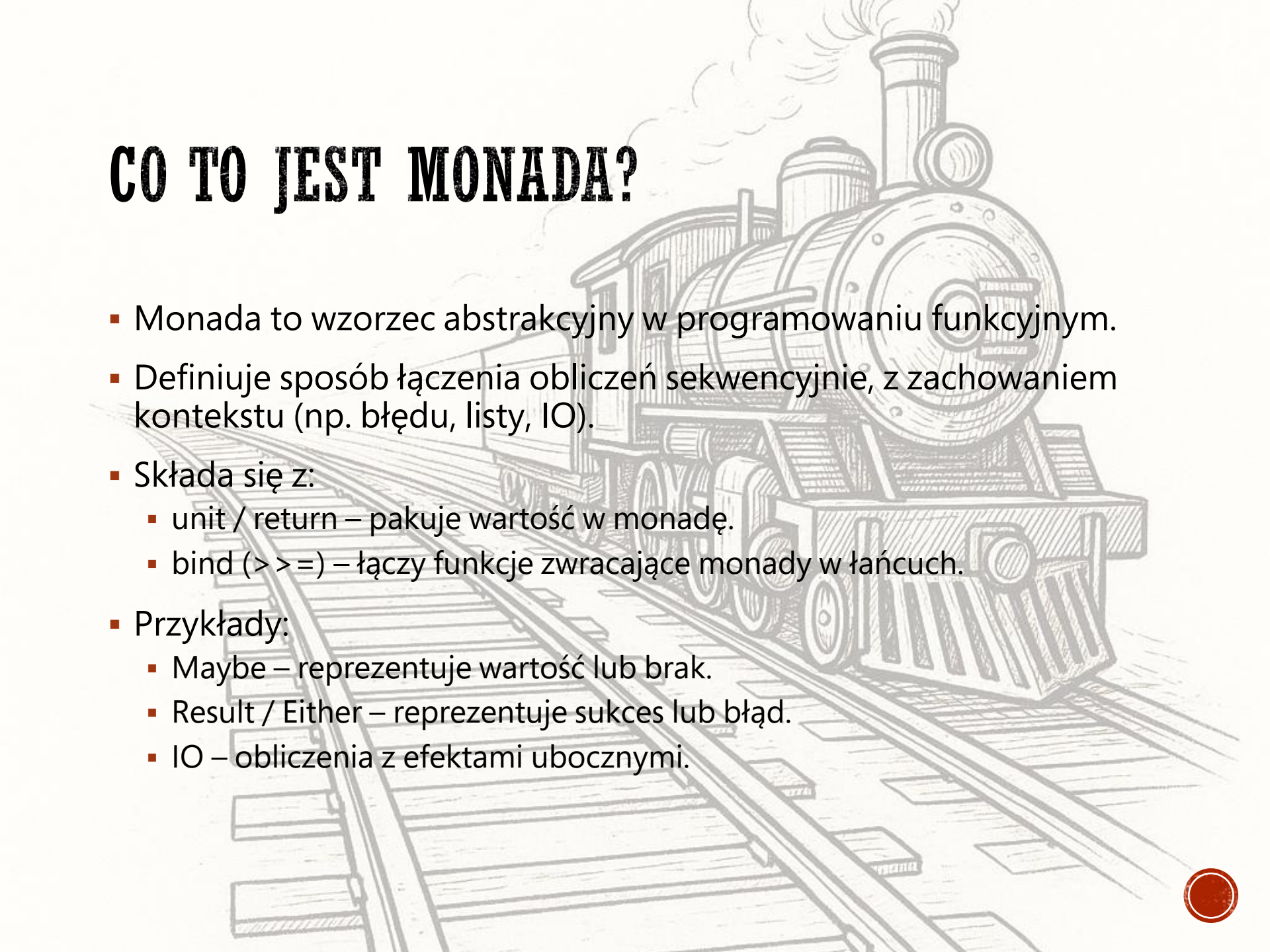
```
divide :: Int -> Int -> Either String Int
divide _ 0 = Left "division by zero"
divide a b = Right (a `div` b)

main = case divide 10 0 of
    Right result -> print result
    Left err -> putStrLn ("Error: " ++ err)
```

NAJBARDZIEJ ELEGANCKIE ROZWIĄZANIE?

- Rust / Haskell / Erlang – bo wymuszają jawne błędy w typach (Result, Either, krotki).
- Go jest prosty, ale powtarzalny („error handling boilerplate”).
- Java / C# mają ukryte wyjątki 👉 mniej przewidywalne.

CO TO JEST MONADA?

- Monada to wzorzec abstrakcyjny w programowaniu funkcyjnym.
- Definiuje sposób łączenia obliczeń sekwencyjnie, z zachowaniem kontekstu (np. błędu, listy, IO).
- Składa się z:
  - unit / return – pakuje wartość w monadę.
  - bind (>>=) – łączy funkcje zwracające monady w łańcuch.
- Przykłady:
  - Maybe – reprezentuje wartość lub brak.
  - Result / Either – reprezentuje sukces lub błąd.
  - IO – obliczenia z efektami ubocznymi.

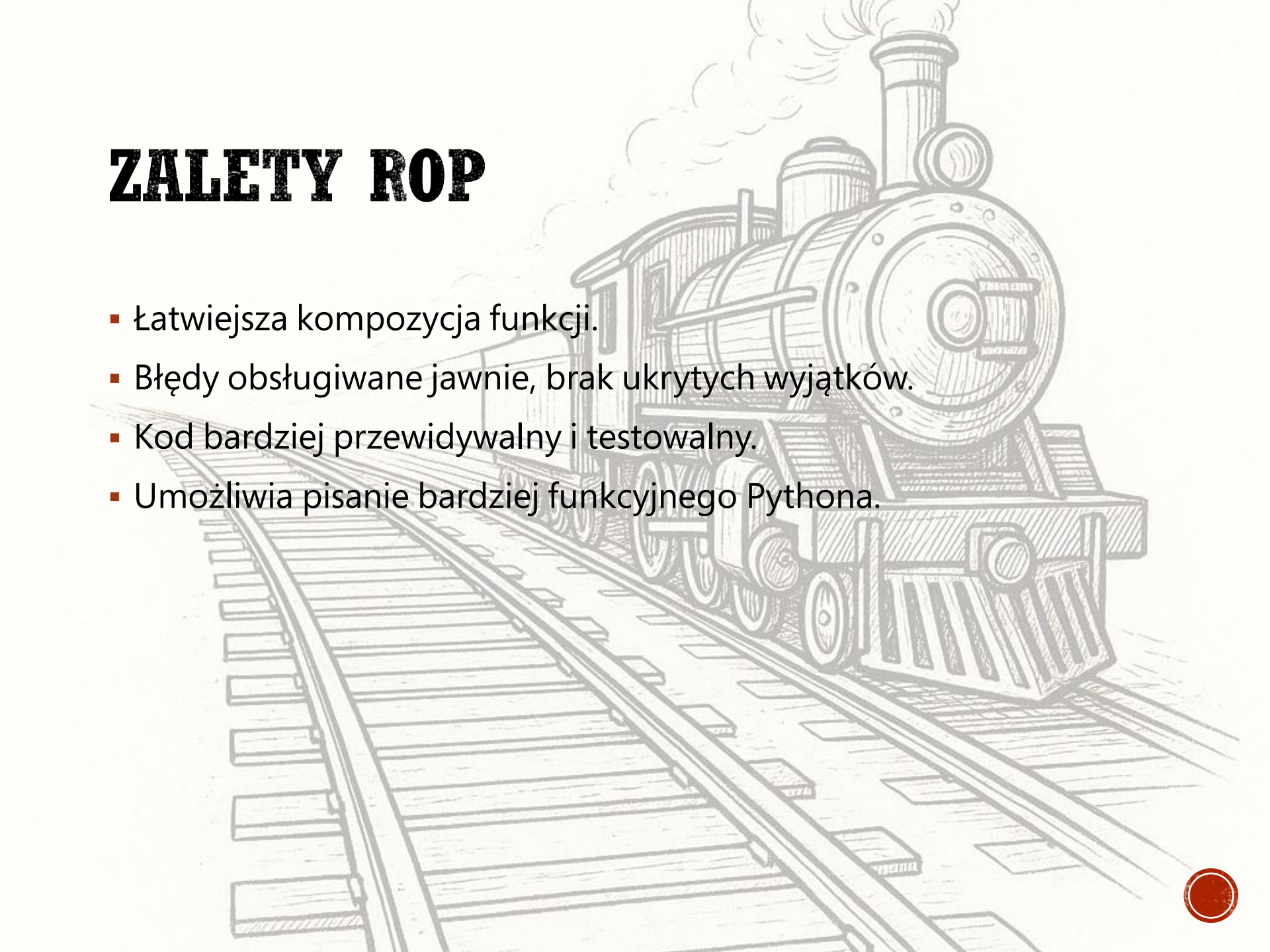


RAILWAY ORIENTED PROGRAMMING (ROP)

- Metafora torów kolejowych
- Dwa tory – jeden dla poprawnych wartości, drugi dla błędów
- Funkcje transformujące – każda funkcja przyjmuje wartość z toru sukcesu i zwraca wynik na tor sukcesu lub przełącza na tor błędu.
- Łańcuchy operacji – funkcje można łączyć (pipe/chain), a przepływ sam wybiera odpowiedni tor.
- Brak wyjątków runtime – zamiast try/except, błędy są jawne i obsługiwane w przepływie danych.
- Kompozycja – logika biznesowa budowana jest z małych funkcji, które można łatwo komponować.
- Deklaratywność – opisujemy co się dzieje, a nie jak łapać wyjątki.
- Przewidywalność – błędy nie „wyskakują” znikąd, są elementem kontraktu funkcji.
- Inspiracja funkcyjna – wzorzec pochodzi z F#, Haskella i monad.

ZALETY ROP

- Łatwiejsza kompozycja funkcji.
- Błędy obsługiwane jawnie, brak ukrytych wyjątków.
- Kod bardziej przewidywalny i testowalny.
- Umożliwia pisanie bardziej funkcyjnego Pythona.



WADY RO

- Większa „ceremonia” w prostych przypadkach.
- Wymaga zmiany sposobu myślenia (nieintuicyjne dla programistów OOP).
- Trudniejsze debugowanie, jeśli ktoś nie zna koncepcji.
- Python nie wspiera tego natywnie, potrzebne biblioteki (np. returns).
- Opakowanie kosztuje 😉

BIBLIOTEKA RETURNS

- Implementuje monadyczne podejście znane z języków funkcyjnych w Pythonie.
- Główne typy:
  - Result[a, b] – sukces (Success) lub błąd (Failure).
  - Maybe[a] – Some lub Nothing.
  - IO, FutureResult, RequiresContext.
- Funkcje: map, bind, alt, unwrap_or, failure(), itd.
- Bezpieczne i przewidywalne API.
- Zgodność z ROP (Railway Oriented Programming).

PRZYKŁAD SYNCHRONICZNY

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

def parse_int(value: str) -> Result[int, str]:
    try:
        return Success(int(value))
    except ValueError:
        return Failure(f"Cannot parse {value}")

result = parse_int("42").map(lambda x: x * 2)
# -> Success(84)

result = parse_int("abc").map(lambda x: x * 2)
# -> Failure("Cannot parse abc")
```

PRZYKŁAD SYNCHRONICZNY

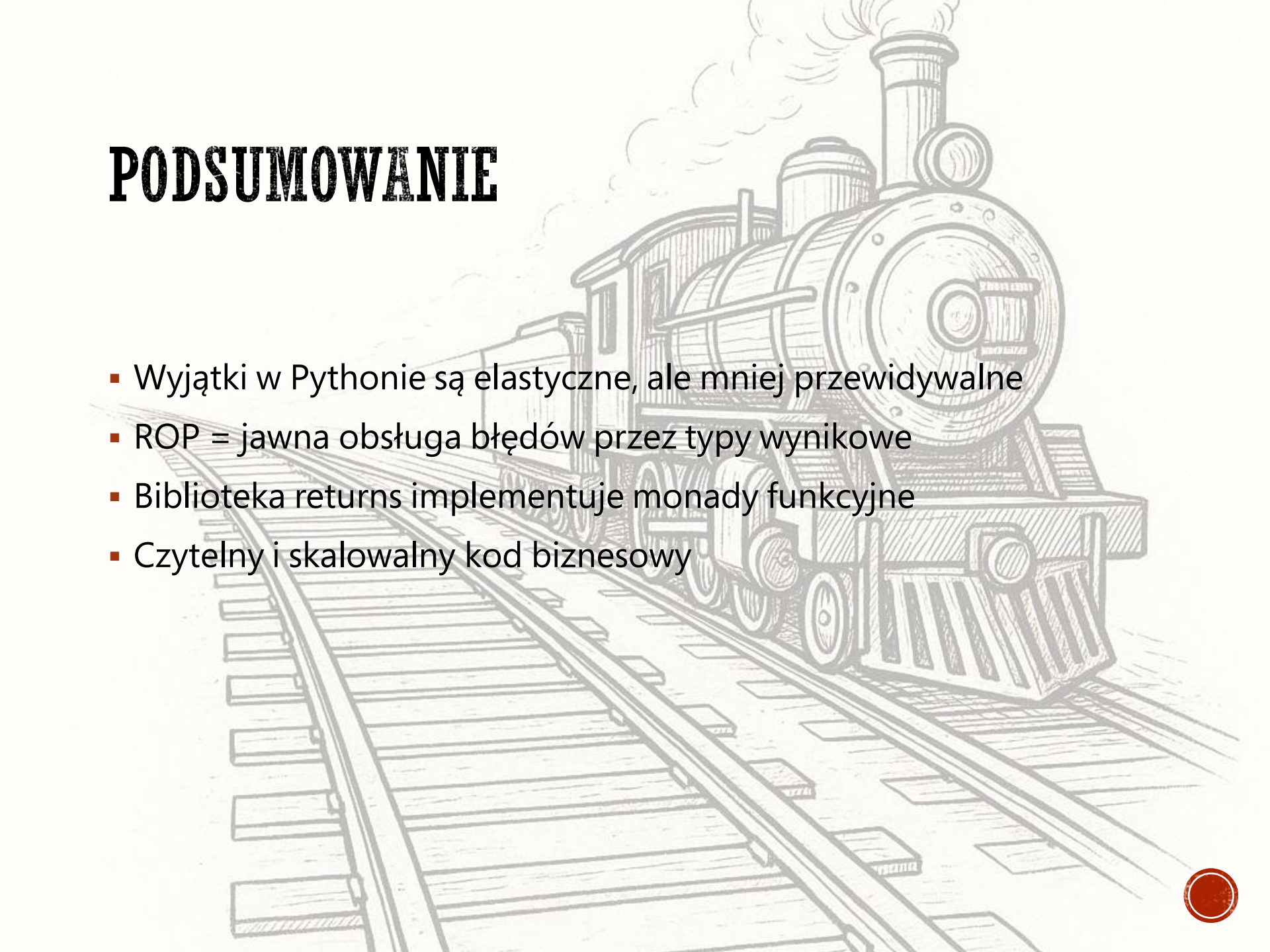
```python
from returns.result import Result, Success, Failure

def validate_age(age: int) -> Result[int, str]:
    return Success(age) if age >= 18 else Failure("Za młody")

def parse_age(value: str) -> Result[int, str]:
    try:
        return Success(int(value))
    except ValueError:
        return Failure("Błędna liczba")

result = (
    parse_age("21")
    .bind(validate_age)
    .map(lambda x: f"Witaj! Wiek: {x}")
)
# -> Success("Welcome! Age: 21")
```

PRZYKŁAD ASYNCHRONICZNY

```python
import aiohttp
from returns.future import FutureResult, future_safe

@future_safe
async def fetch_data(url: str) -> str:
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()

async def main():
    result = await fetch_data("https://example.com")
    # result to Success(...) lub Failure(...)
    final = result.map(lambda x: len(x))
    print(final)
```

PODSUMOWANIE

- Wyjątki w Pythonie są elastyczne, ale mniej przewidywalne
- ROP = jawna obsługa błędów przez typy wynikowe
- Biblioteka returns implementuje monady funkcyjne
- Czytelny i skalowalny kod biznesowy



Q&A...

DZIĘKUJĘ ZA UWAGĘ 😄

Psss, kod źródłowy 👌