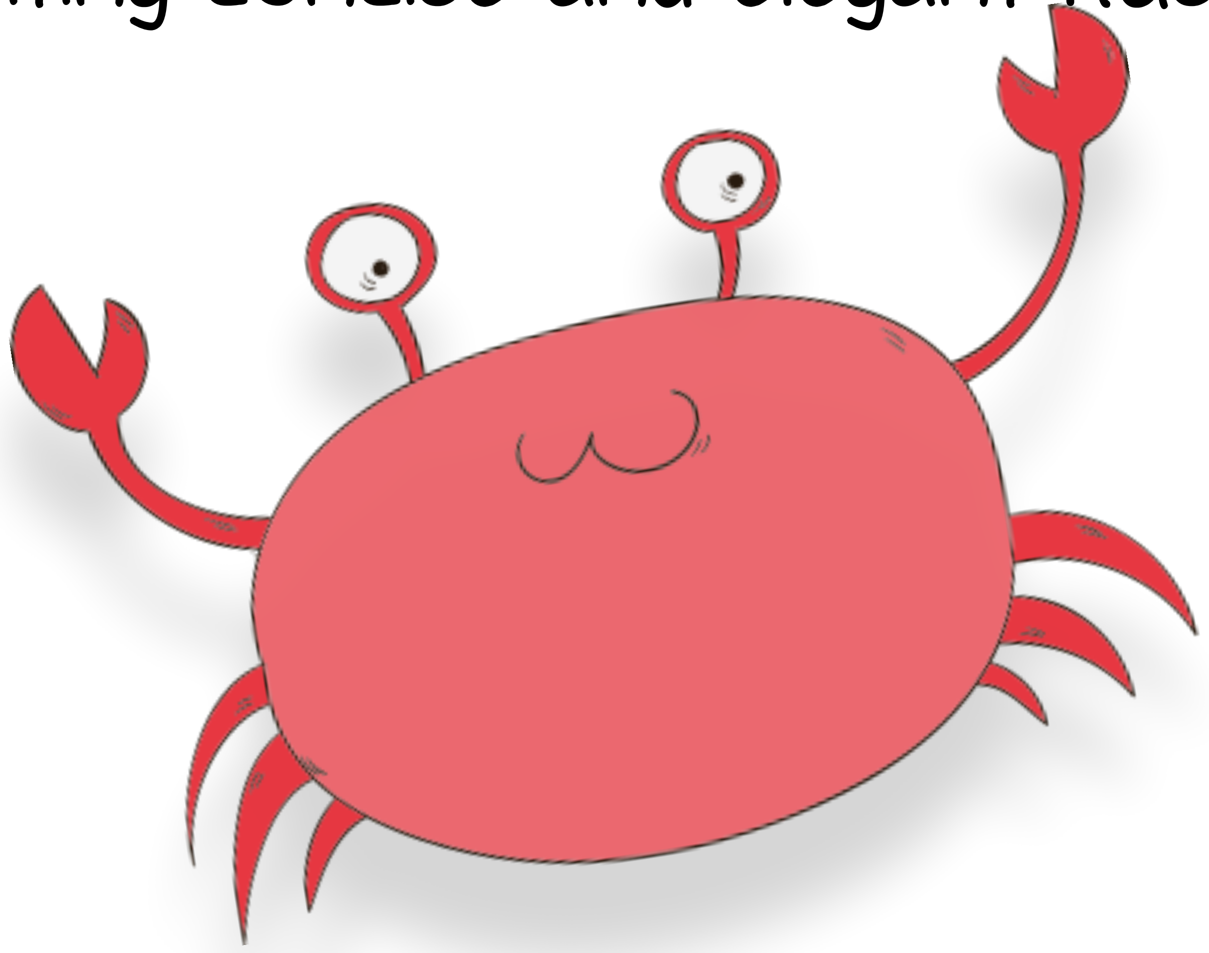
Idiomatic Rust

Writing concise and elegant Rust code



Matthias Endler

- Düsseldorf, Germany
- Backend Engineer at trivago
- Website performance
- Hot Chocolate

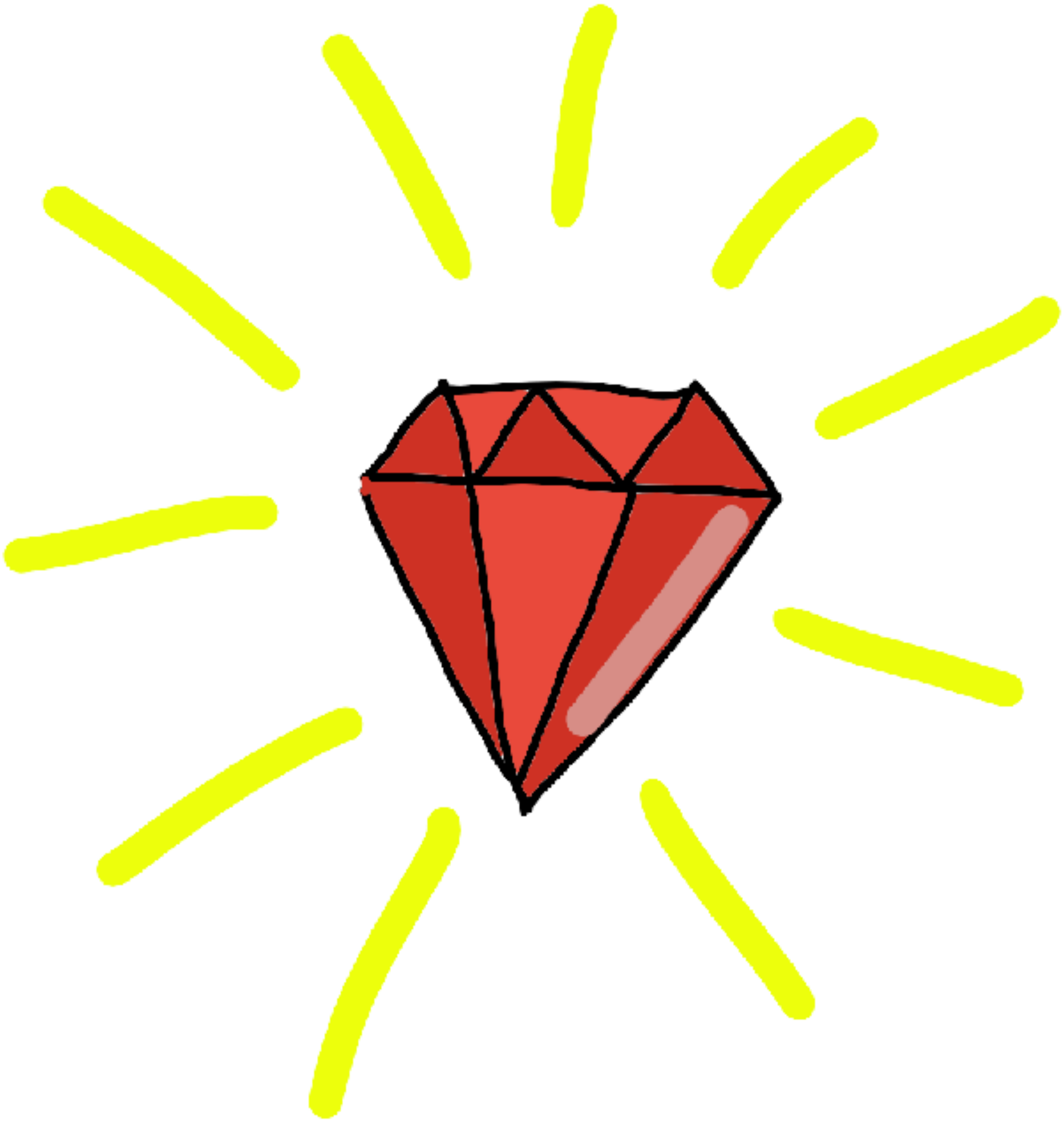
matthiasendler

mre

matthias-endler.de

EXPECTATION...

REALITY...



Python

The Zen of Python

Image: Monty Python and the Holy Grail (1975)

Zen of Python

What is idiomatic Rust?

What is idiomatic?

The most
concise, convenient and common
way of accomplishing a task
in a programming language.

Tim Mansfield

```
public bool IsTrue(bool b)
{
   if (b == true)
   {
      return true;
   }
   return false;
}
```

http://codecrap.com/content/172/

Idiomatic Rust

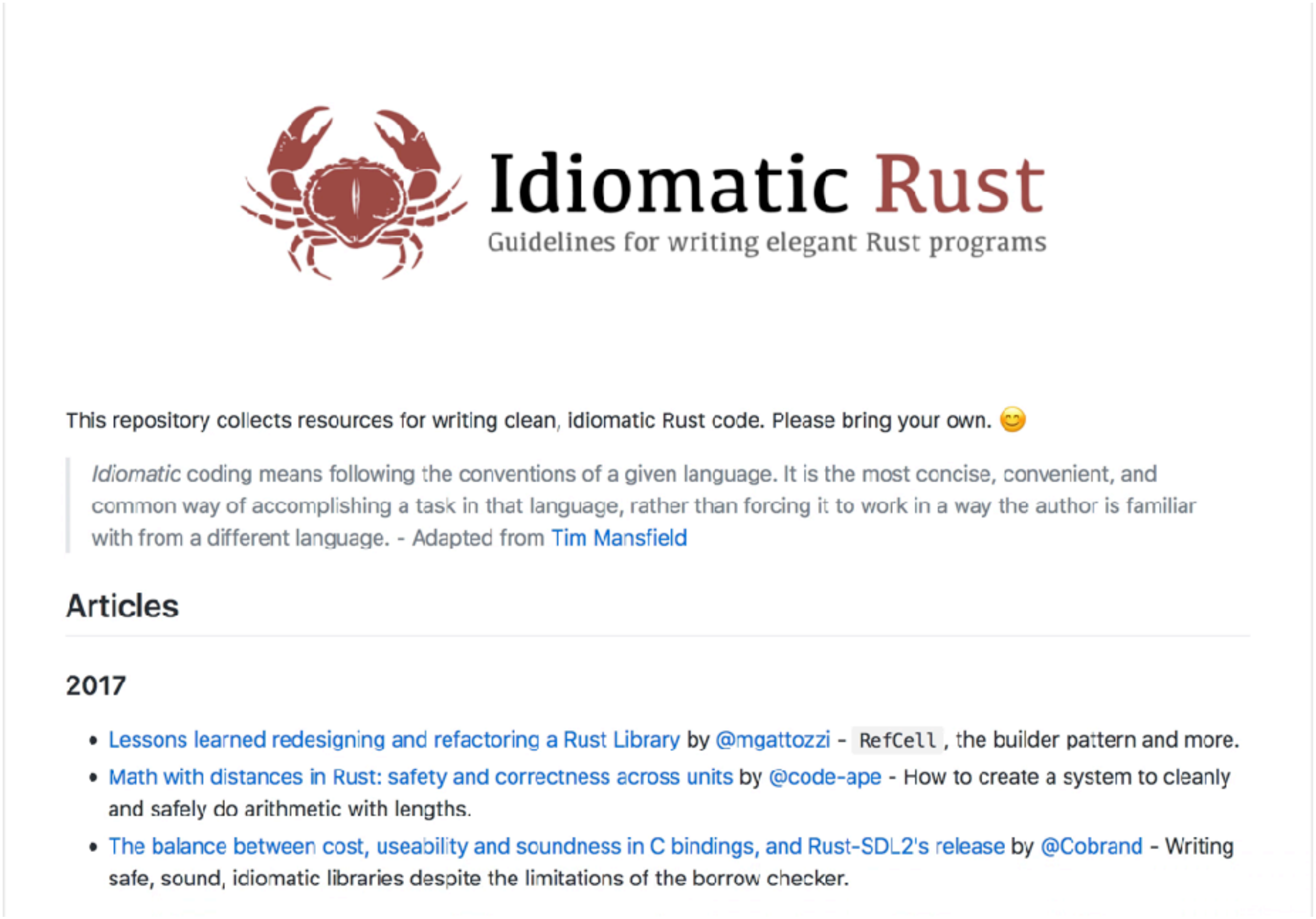
syntax

semantics

design patterns

Idiomatic Rust

$$\begin{tikzcd}
\text{syntax} \arrow[r] & \text{use rustfmt} \\
\text{semantics} \arrow[r] & \text{???} \\
\text{design patterns} \arrow[r] & \text{rust-unofficial/patterns}
\end{tikzcd}$$



Idiomatic Rust

Guidelines for writing elegant Rust programs

This repository collects resources for writing clean, idiomatic Rust code. Please bring your own. 😊

> *Idiomatic* coding means following the conventions of a given language. It is the most concise, convenient, and common way of accomplishing a task in that language, rather than forcing it to work in a way the author is familiar with from a different language. - Adapted from Tim Mansfield

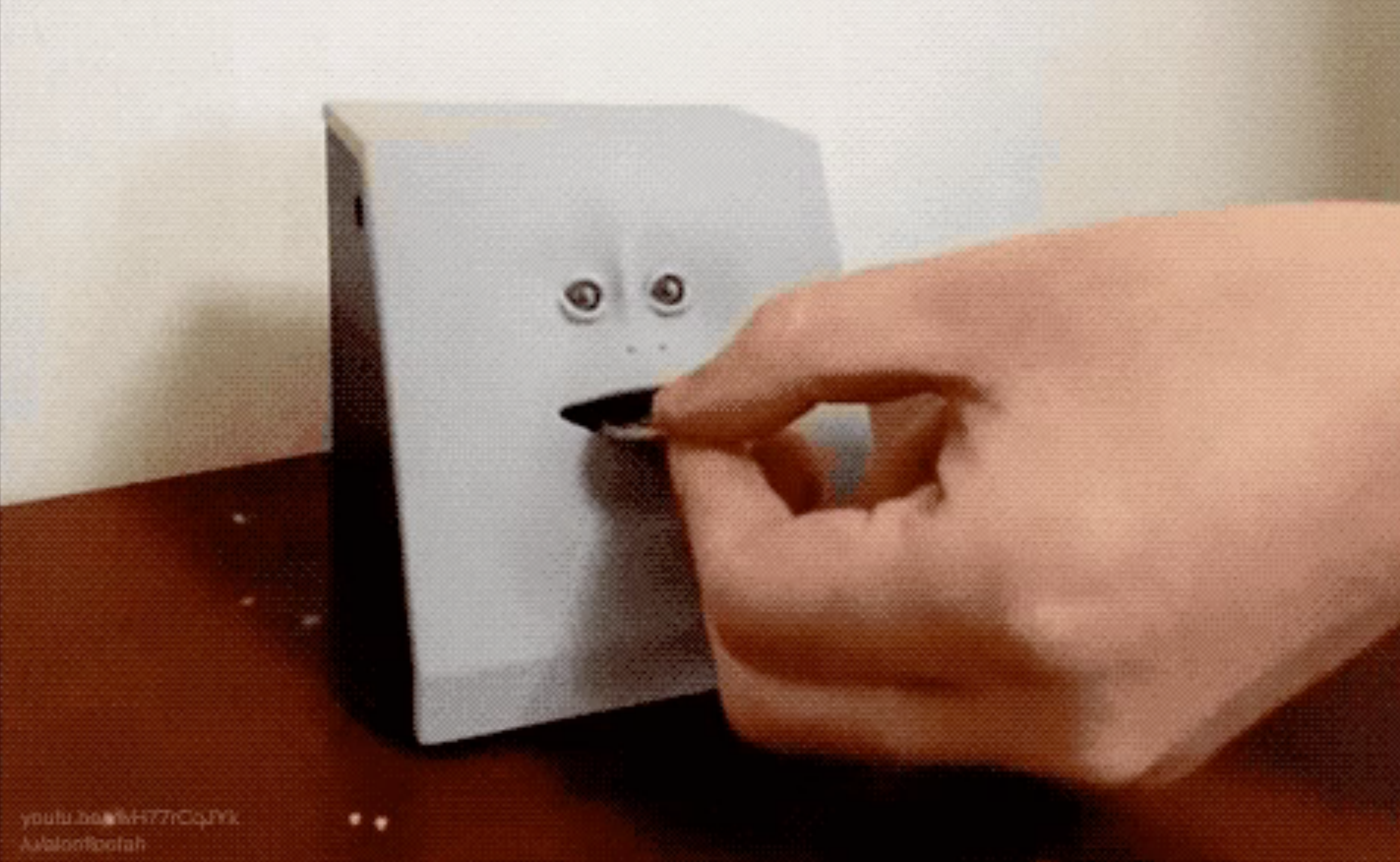
Articles

2017

- Lessons learned redesigning and refactoring a Rust Library by @mgattozzi - `RefCell`, the builder pattern and more.
- Math with distances in Rust: safety and correctness across units by @code-ape - How to create a system to cleanly and safely do arithmetic with lengths.
- The balance between cost, useability and soundness in C bindings, and Rust-SDL2's release by @Cobrand - Writing safe, sound, idiomatic libraries despite the limitations of the borrow checker.

https://github.com/mre/idiomatic-rust

Case study: Handling money in Rust



Task:

Parse money, e.g.
20,42 Dollar or 140 Euro.

```rust
fn parse_money(input: &str) {



    // TODO



}
```

```rust
fn parse_money(input: &str) -> (i32, String) {







}
```

“magic“ error constants

```rust
fn parse_money(input: &str) -> (i32, String) {
    let parts: Vec<&str> = input.split_whitespace().collect();
    let maybe_amount = parts[0].parse();
    if maybe_amount.is_err() {
        return (-1, "invalid".to_string());
    }
    let currency = parts[1].to_string();
    return (maybe_amount.unwrap(), currency);
}
```

Use unwrap()

```
fn parse_money(input: &str) -> (i32, String) {
    let parts: Vec<&str> = input.split_whitespace().collect();
    let amount = parts[0].parse().unwrap();
    let currency = parts[1].to_string();
    return (amount, currency);
}
```

```
parse_money("140 Euro");
(140, "Euro")
```

```
parse_money("140.01 Euro");
```

```
thread 'main' panicked at 'called `Result::unwrap()`
on an `Err` value: ParseIntError { kind: InvalidDigit
}', src/libcore/result.rs:906:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

unwrap will panic on error

```rust
fn parse_money(input: &str) -> (i32, String) {
    let parts: Vec<&str> = input.split_whitespace().collect();
    let amount = parts[0].parse().unwrap();
    let currency = parts[1].to_string();
    return (amount, currency);
}
```

replace unwrap with ?

```rust
fn parse_money(input: &str) -> Result<(i32, String), ParseIntError> {
    let parts: Vec<&str> = input.split_whitespace().collect();
    let amount = parts[0].parse()?;
    let currency = parts[1].to_string();
    return Ok((amount, currency));
}
```

Bro blem?

```
parse_money("140.01 Euro");
```

```
Err(ParseIntError { kind: InvalidDigit })
```

Wrong type for parse()

```
fn parse_money(input: &str) -> Result<(i32, String), ParseIntError> {
    let parts: Vec<&str> = input.split_whitespace().collect();
    let amount = parts[0].parse()?;
    let currency = parts[1].to_string();
    return Ok((amount, currency));
}
```

use float

```rust
fn parse_money(input: &str) -> Result<(f32, String), ParseFloatError> {
    let parts: Vec<&str> = input.split_whitespace().collect();
    let amount = parts[0].parse()?;
    let currency = parts[1].to_string();
    return Ok((amount, currency));
}
```

Don't use float for real-world money objects!

```
parse_money("140.01 Euro");
Ok((140.01, "Euro"))
```

```
parse_money("140.01");
```

```
thread 'main' panicked at 'index out of bounds: the
len is 1 but the index is 1', /Users/travis/build/
rust-lang/rust/src/liballoc/vec.rs:1551:10
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Unchecked vector index

```rust
fn parse_money(input: &str) -> Result<(f32, String), ParseFloatError> {
    let parts: Vec<&str> = input.split_whitespace().collect();
    let amount = parts[0].parse()?;
    let currency = parts[1].to_string();
    return Ok((amount, currency));
}
```

use custom error

```rust
fn parse_money(input: &str) -> Result<(f32, String), MoneyError> {
    let parts: Vec<&str> = input.split_whitespace().collect();
    if parts.len() != 2 {
        Err(MoneyError::ParseError)
    } else {
        let (amount, currency) = (parts[0], parts[1]);
        Ok((amount.parse()?, currency.to_string()))
    }
}
```

```rust
#[derive(Debug)]
pub enum MoneyError {
    ParseError,
}
```

```rust
#[derive(Debug, Fail)]
enum MoneyError {
    #[fail(display = "Invalid input: {}", _0)]
    ParseAmount(ParseFloatError),

    #[fail(display = "{}", _0)]
    ParseFormatting(String),
}

impl From<ParseFloatError> for MoneyError {
    fn from(e: ParseFloatError) -> Self {
        MoneyError::ParseAmount(e)
    }
}
```

https://github.com/withoutboats/failure

```
println!("{:?}", parse_money("140.01"));

Err(ParseFormatting("Expecting amount and currency"))

println!("{:?}", parse_money("OneMillion Euro"));

Err(ParseAmount(ParseFloatError { kind: Invalid }))

println!("{:?}", parse_money("100 Euro"));

Ok((100, "Euro"))
```

explicit length check

```rust
fn parse_money(input: &str) -> Result<(f32, String), MoneyError> {
    let parts: Vec<&str> = input.split_whitespace().collect();
    if parts.len() != 2 {
        Err(MoneyError::ParseFormatting(
            "Expecting amount and currency".into(),
        ))
    } else {
        let (amount, currency) = (parts[0], parts[1]);
        Ok((amount.parse()?, currency.to_string()))
    }
}
```

slice patterns

```rust
#![feature(slice_patterns)]

fn parse_money(input: &str) -> Result<(f32, String), MoneyError> {
    let parts: Vec<&str> = input.split_whitespace().collect();

    match parts[..] {
        [amount, currency] => Ok((amount.parse()?, currency.to_string())),
        _ => Err(MoneyError::ParseFormatting(
            "Expecting amount and currency".into(),
        )),
    }
}
```

use own type for money

```
#![feature(slice_patterns)]

fn parse_money(input: &str) -> Result<Money, MoneyError> {
    let parts: Vec<&str> = input.split_whitespace().collect();

    match parts[..] {
        [amount, curr] => Ok(Money::new(amount.parse()?, curr.parse()?)),
        _ => Err(MoneyError::ParseFormatting(
            "Expecting amount and currency".into(),
        )),
    }
}
```

use own type for money

```rust
#[derive(Debug)]
struct Money {
    amount: f32,
    currency: Currency,
}

impl Money {
    fn new(amount: f32, currency: Currency) -> Self {
        Money { amount, currency }
    }
}
```

use own type for money

```rust
#[derive(Debug)]
enum Currency {
    Dollar,
    Euro,
}

impl std::str::FromStr for Currency {
    type Err = MoneyError;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        match s.to_lowercase().as_ref() {
            "dollar" | "$" => Ok(Currency::Dollar),
            "euro" | "eur" | "€" => Ok(Currency::Euro),
            _ => Err(MoneyError::ParseCurrency("Unknown currency".into())),
        }
    }
}
```

use own type for money

```rust
impl std::str::FromStr for Money {
    type Err = MoneyError;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        let parts: Vec<&str> = s.split_whitespace().collect();

        match parts[..] {
            [amount, currency] => Ok(Money::new(amount.parse()?, currency.parse()?)),
            _ => Err(MoneyError::ParseFormatting(
                "Expecting amount and currency".into(),
            )),
        }
    }
}
```

```
"140.01".parse::<Money>()
Err(ParseFormatting("Expecting amount and currency"))

"OneMillion Bitcoin".parse::<Money>()
Err(ParseAmount(ParseFloatError { kind: Invalid }))

"100 €".parse::<Money>()
Ok(Money { amount: 100.0, currency: Euro })

"42.24 Dollar".parse::<Money>()
Ok(Money { amount: 42.42, currency: Dollar })
```

Thank you!

matthias-endler.de

github.com/mre/idiomatic-rust

...use clippy!

Crab title image designed by freepik