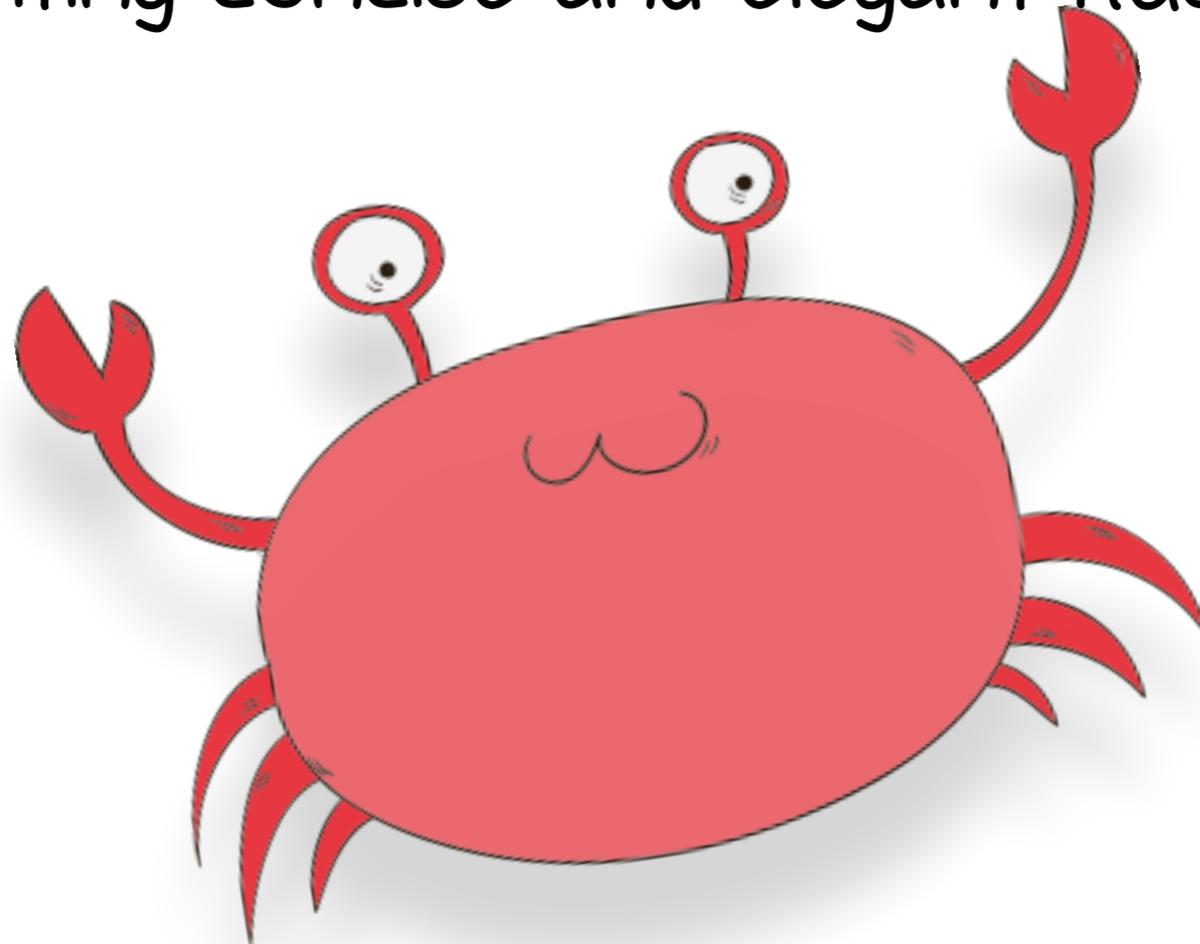


# Idiomatic Rust

Writing concise and elegant Rust code



# Matthias Ender



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



matthiasender



mre



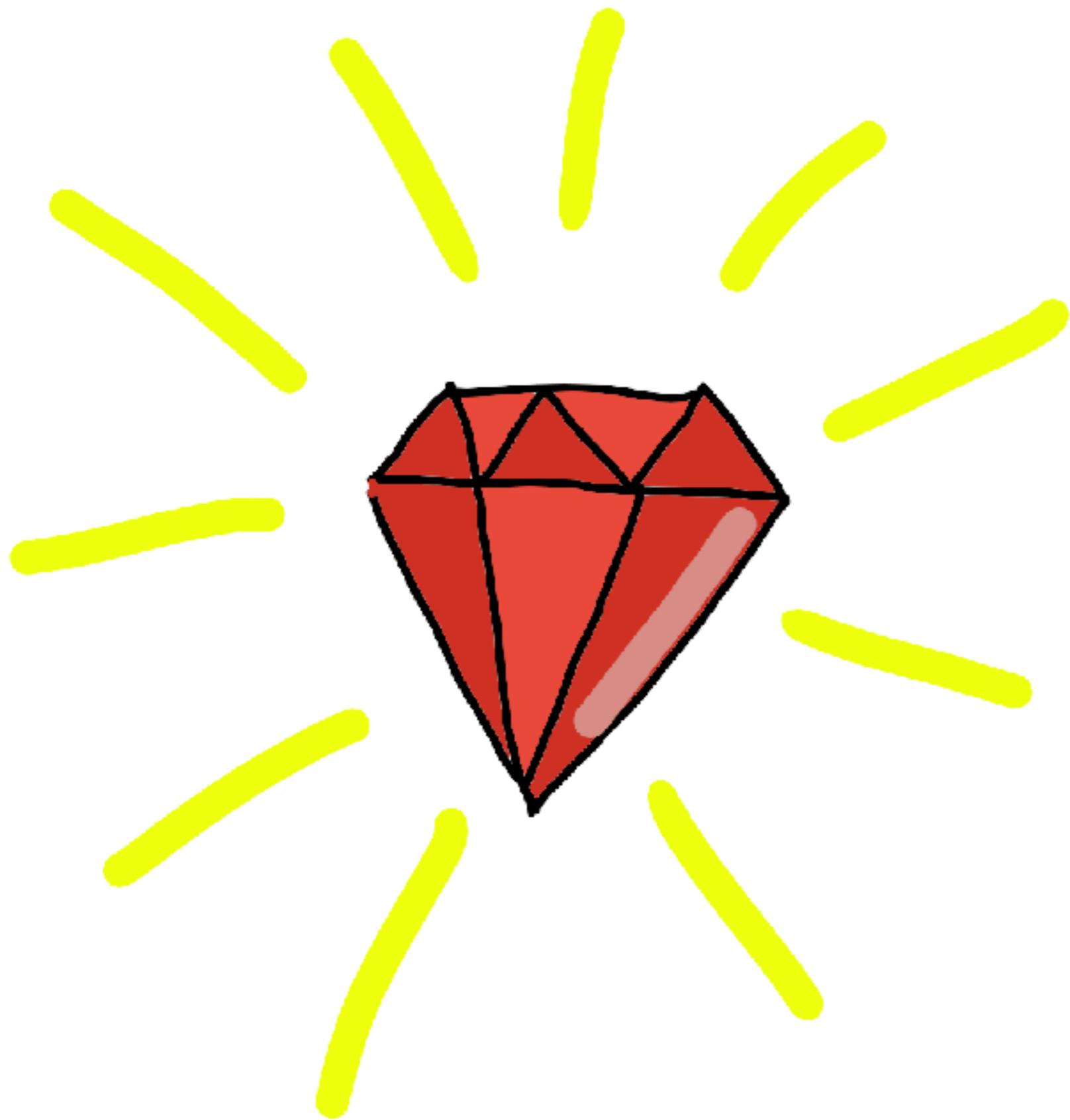
[matthias-ender.de](http://matthias-ender.de)

**EXPECTATION...**



**REALITY...**





Python

# The Zen of Python



**Image: Monty Python and the Holy Grail (1975)**

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;
}
```

# Idiomatic Rust

syntax

semantics

design patterns

# Idiomatic Rust

syntax  use rustfmt

semantics  ???

design patterns  rust-unofficial/patterns





# 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

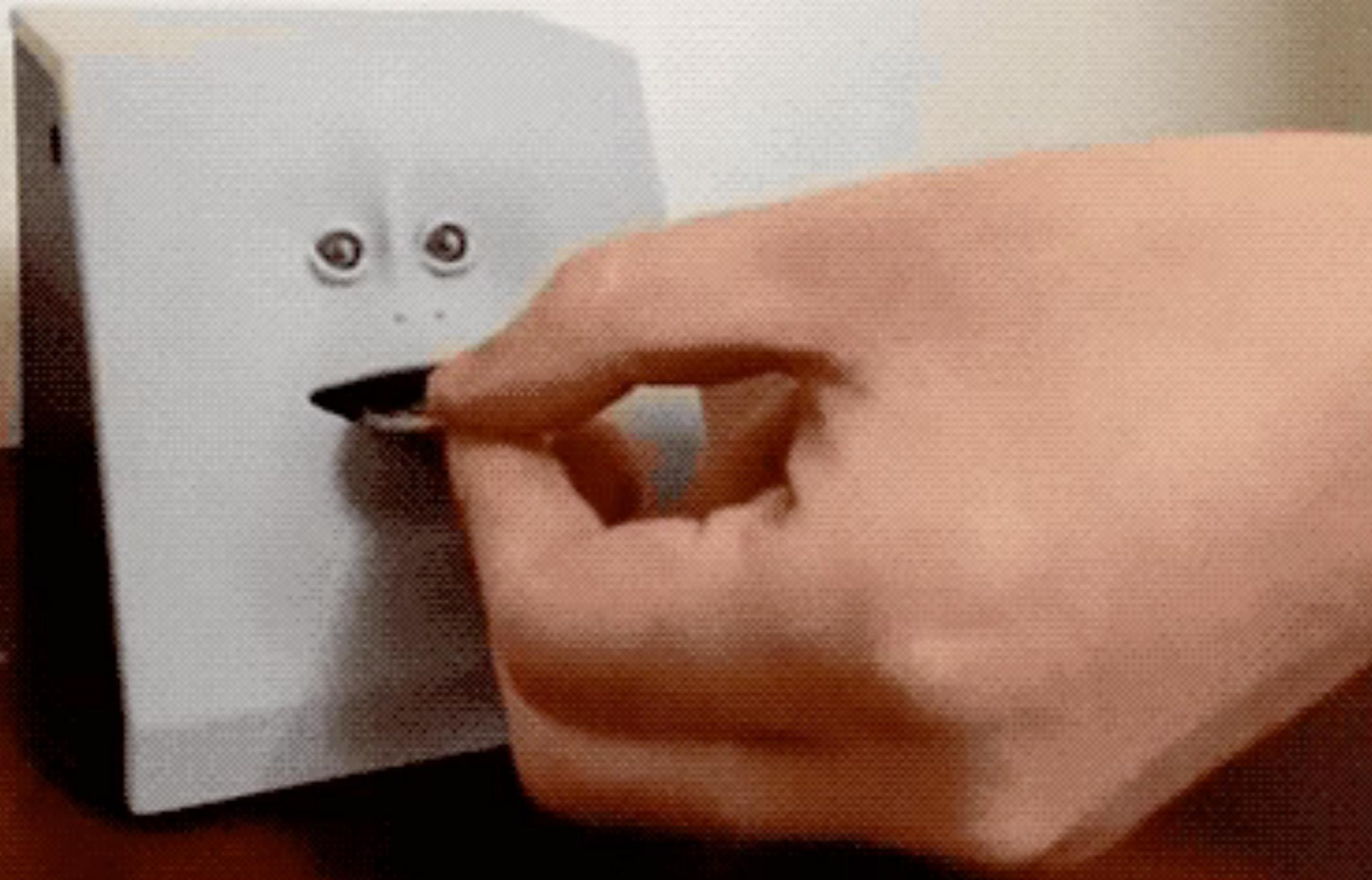
---

### 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.

```
1 fn parse_money(input: &str) {  
2  
3  
4  
5     // TODO  
6  
7  
8  
9 }
```

```
1 fn parse_money(input: &str) -> (i32, String) {  
2  
3  
4  
5  
6  
7  
8  
9 }
```

## "magic" error constants

```
1 fn parse_money(input: &str) -> (i32, String) {
2     let parts: Vec<&str> = input.split_whitespace().collect();
3     let maybe_amount = parts[0].parse();
4     if maybe_amount.is_err() {
5         return (-1, "invalid".to_string());
6     }
7     let currency = parts[1].to_string();
8     return (maybe_amount.unwrap(), currency);
9 }
```

use unwrap()

```
1 fn parse_money(input: &str) -> (i32, String) {  
2     let parts: Vec<&str> = input.split_whitespace().collect();  
3     let amount = parts[0].parse().unwrap();  
4     let currency = parts[1].to_string();  
5     return (amount, currency);  
6 }
```

```
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

```
1 fn parse_money(input: &str) -> (i32, String) {  
2     let parts: Vec<&str> = input.split_whitespace().collect();  
3     let amount = parts[0].parse().unwrap();  
4     let currency = parts[1].to_string();  
5     return (amount, currency);  
6 }
```

replace unwrap with ?

```
1 fn parse_money(input: &str) -> Result<i32, String>, ParseIntError>
2     let parts: Vec<&str> = input.split_whitespace().collect();
3     let amount = parts[0].parse()?;
4     let currency = parts[1].to_string();
5     return Ok(amount, currency);
6 }
```



**Bro blem?**

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

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

# Wrong type for parse()

```
1 fn parse_money(input: &str) -> Result<(i32, String), ParseIntError> {  
2     let parts: Vec<&str> = input.split_whitespace().collect();  
3     let amount = parts[0].parse()?;  
4     let currency = parts[1].to_string();  
5     return Ok((amount, currency));  
6 }
```

use float

```
1 fn parse_money(input: &str) -> Result<(f32, String), ParseFloatError> {  
2     let parts: Vec<&str> = input.split_whitespace().collect();  
3     let amount = parts[0].parse()?;  
4     let currency = parts[1].to_string();  
5     return Ok((amount, currency));  
6 }
```

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



Using float for real-world money objects...

use float

```
1 fn parse_money(input: &str) -> Result<(f32, String), ParseFloatError> {  
2     let parts: Vec<&str> = input.split_whitespace().collect();  
3     let amount = parts[0].parse()?;  
4     let currency = parts[1].to_string();  
5     return Ok((amount, currency));  
6 }
```

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

```
1 fn parse_money(input: &str) -> Result<(f32, String), ParseFloatError> {  
2     let parts: Vec<&str> = input.split_whitespace().collect();  
3     let amount = parts[0].parse()?;  
4     let currency = parts[1].to_string();  
5     return Ok((amount, currency));  
6 }
```

## use custom error

```
1 fn parse_money(input: &str) -> Result<(f32, String), MoneyError> {
2     let parts: Vec<&str> = input.split_whitespace().collect();
3     if parts.len() != 2 {
4         Err(MoneyError::ParseError)
5     } else {
6         let (amount, currency) = (parts[0], parts[1]);
7         Ok((amount.parse()?, currency.to_string()))
8     }
9 }
```

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

```

#[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)
    }
}

```

```
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

```
1 fn parse_money(input: &str) -> Result<(f32, String), MoneyError> {
2     let parts: Vec<&str> = input.split_whitespace().collect();
3     if parts.len() != 2 {
4         Err(MoneyError::ParseFormatting(
5             "Expecting amount and currency".into(),
6             ))
7     } else {
8         let (amount, currency) = (parts[0], parts[1])
9         Ok((amount.parse()?, currency.to_string()))
10    }
11 }
```

# slice patterns

```
#![feature(slice_patterns)]

1 fn parse_money(input: &str) -> Result<(f32, String), MoneyError> {
2     let parts: Vec<&str> = input.split_whitespace().collect();
3
4     match parts[..] {
5         [amount, currency] => Ok((amount.parse()?, currency.to_string())),
6         _ => Err(MoneyError::ParseFormatting(
7             "Expecting amount and currency".into(),
8         )),
9     }
10 }
```

# use own type for money

```
#![feature(slice_patterns)]

1 fn parse_money(input: &str) -> Result<Money, MoneyError> {
2     let parts: Vec<&str> = input.split_whitespace().collect();
3
4     match parts[..] {
5         [amount, curr] => Ok(Money::new amount.parse()?, curr.parse()?)),
6         _ => Err(MoneyError::ParseFormatting(
7             "Expecting amount and currency".into(),
8         )),
9     }
10 }
```

use own type for money

```
#[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

```
1 #[derive(Debug)]
2 enum Currency {
3     Dollar,
4     Euro,
5 }
6
7 impl std::str::FromStr for Currency {
8     type Err = MoneyError;
9
10    fn from_str(s: &str) -> Result<Self, Self::Err> {
11        match s.to_lowercase().as_ref() {
12            "dollar" | "$" => Ok(Currency::Dollar),
13            "euro" | "eur" | "€" => Ok(Currency::Euro),
14            _ => Err(MoneyError::ParseCurrency("Unknown currency".into())),
15        }
16    }
17 }
```

# use own type for money

```
1  impl std::str::FromStr for Money {
2      type Err = MoneyError;
3
4      fn from_str(s: &str) -> Result<Self, Self::Err> {
5          let parts: Vec<&str> = s.split_whitespace().collect();
6
7          match parts[..] {
8              [amount, curr] => Ok(Money::new(amount.parse()?, curr.parse()?)),
9              _ => Err(MoneyError::ParseFormatting(
10                 "Expecting amount and currency".into(),
11                 )),
12          }
13      }
14 }
```



```
"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: 42.24, currency: Dollar })
```

Thank you!

[matthias-endler.de](http://matthias-endler.de)

[@matthiasendler](https://twitter.com/matthiasendler)

[github.com/mre/idiomatic-rust](https://github.com/mre/idiomatic-rust)

...use the [clippy crate!](#)



Crab title image designed by freepik