I'm a Python programmer — Should I bother learning Rust?

Christian Kauhaus
Python User Group Berlin
2020-07-09

Christian Kauhaus

- Systems Engineer @
  Flying Circus Internet Operations GmbH, Halle/Saale
- Programming in Rust since 2016
- kc@flyingcircus.io
- Twitter, GitHub etc: **@ckauhaus**

Agenda

1. Language comparison

2. Rust language safari

3. Interactive code example

Language comparison

—or—

What attracted me to Rust?

What attracted me to Rust

- Powerful static type system
- Memory safety
- No garbage collection
- Performance comparable to C/C++
- No runtime, easy to embed
- Static binaries
- Low level control

Discoveries on the way (cool)

- Easy and safe parallelization/multi-threading
- Code generation macros
- Systematic error handling
- Multi-paradigm programming
- No OO inheritance
- Abstractions with no/minimal overhead
- Excellent compiler errors
- `cargo` build tooling
- Many high-quality crates (libraries)

Discoveries on the way (not so cool)

- Borrow checker hard to get into
- High language complexity
- Need to master a large subset to become productive
- Long compile times
- No REPL
- (Relative) source code verbosity
- Standard library (relatively) sparse

Rust and Python complement each other

Rust is strong at

- Performance
- Safety/reliability
- Parallelization
- Low-level resource control

Python is strong at

- Glueing components together
- Scripting
- Easy to change business logic
- Prototyping/exploratory programming

High-level comparison

Performance

10-100 times faster

Source code size

2—4 times larger

Run-time reliability

Much higher

High-level comparison

Concurrent high-performance programming

Possible at all

Bare metal access

Possible at all

Rust language safari

—or—

Top sights for Pythonistas

Rust has a strong type system

- Strict checking on compile time
- High expressiveness
- Invalid stuff should be impossible to express

Never get things like:

```
AttributeError: 'NoneType' object has no attribute 'close'
```

Rust has explicit mutability control

Does work:

```
fn main() {
    let mut hello = String::from("Hello");
    hello += " world!";
    println!("{}", hello);
}
```

Does not work:

```
fn main() {
    let hello = "Hello"; // read-only string slice &str
    hello += " world!";
    println!("{}", hello);
}
```

Rust has no duck typing

But compile-time polymorphism:

```
impl<K, V> HashMap<K, V>
```

- Gets substituted with actual types at compile time
- Zero run time overhead

```
let mut dict: HashMap<u64, String> = HashMap::new();
```

Rust has no »None«

Nullable types are represented as Option:

```
enum Option<T> {
    Some(T),
    None
}
```

From HashMap @ stdlib

```
impl HashMap<K, V> {
  fn get<K>(&self, key: &K) -> Option<&V>
}
```

Rust has no decorators — but derive macros

Powerful code generation without run time overhead

```
#[derive(Debug, Clone)]
struct Person {
  name: String,
  age: u32
}

/* … */
println!("{:?}", person); // automatic debug formatting
p2 = p1.clone(); // automatic copy constructor
```

Rust has no garbage collector

- Automatic memory management like C++
- Borrow/Livetimes concept
- Object ownership

Memory management rules:

1. Only one scope owns an object
2. Multiple shared (read-only) or exactly one exclusive references may be passed out
3. Data is freed if it falls out of scope

Data access

Move: ownership is transferred

```
impl Connection {
  fn close(self) { /* … */ }
}
```

Borrow: shared, read-only reference

```
impl Path {
  fn join(&self, tail: &Path) -> PathBuf { /* … */ }
}
```

Mutable borrow: exclusive, read-write reference

```
trait io::Read {
  fn read(&mut self, buffer: &mut [u8]) -> Result<u8, io::Error> { /* … */ }
}
```

Ownership

```
use std::collections::HashMap;

#[derive(Debug)]
struct Product {
    name: String,
    vendor: String,
}

fn main() {
    let mut products: HashMap<i64, Product> = HashMap::new();
    let shirt = Product {
        name: "Python polo shirt".to_owned(),
        vendor: "HELLOTUX".to_owned(),
    };
    println!("{:?}", &shirt);
    products.insert(24742, shirt);
    // shift transferred into products hashmap
}
```

Ownership

```
use std::collections::HashMap;

#[derive(Debug)]
struct Product {
    name: String,
    vendor: String,
}

fn main() {
    let mut products = HashMap::new();
    let shirt = Product {
        name: "Python polo shirt".to_owned(),
        vendor: "HELLOTUX".to_owned(),
    };
    products.insert(24742, shirt);

    println!("{:?}", &shirt); // ERROR
}
```

Borrow checking

Does not work:

```
let mut h = HashMap::new();

// populate hashmap
for i in 0..10 {
  h.insert(i, i + 1);
}

// delete every 3rd key
for key in h.keys() {
  if key % 3 == 0 {
    h.remove(key)
  }
}
```

Borrow checking

Compiler error looks like this:

```
   Compiling hashmap v0.1.0
error[E0502]: cannot borrow 'h' as mutable because it is also borrowed as immutable
  --> src/main.rs:10:13
   |
8  |     for key in h.keys() {
   |                --------
   |                |
   |                immutable borrow occurs here
   |                immutable borrow later used here
9  |         if key % 3 == 0 {
10 |             h.remove(key);
   |             ^^^^^^^^^^^^ mutable borrow occurs here
```

Rust has no exceptions — but Results

Result type for fallible execution

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

Example:

```
impl File {
  fn open(path: &Path) -> Result<File, std::io::Error> { /* … */ }
}
```

Rust has interfaces — Traits

Traits introduce flexibility to the type system

```
struct BufReader<R: Read> { /* … */ }

impl<R: Read> BufReader<R> {
  fn new(raw_reader: R) -> BufReader<R> { /* … */ }
}
```

Example:

```
let f = File::open("server.log")?;
let r = BufReader::new(f);
for line in r.lines() {              // lines() not defined for File
  /* … */
}
```

Trait objects

Forget about the concrete type

Dynamic dispatch: types and methods selected at run time

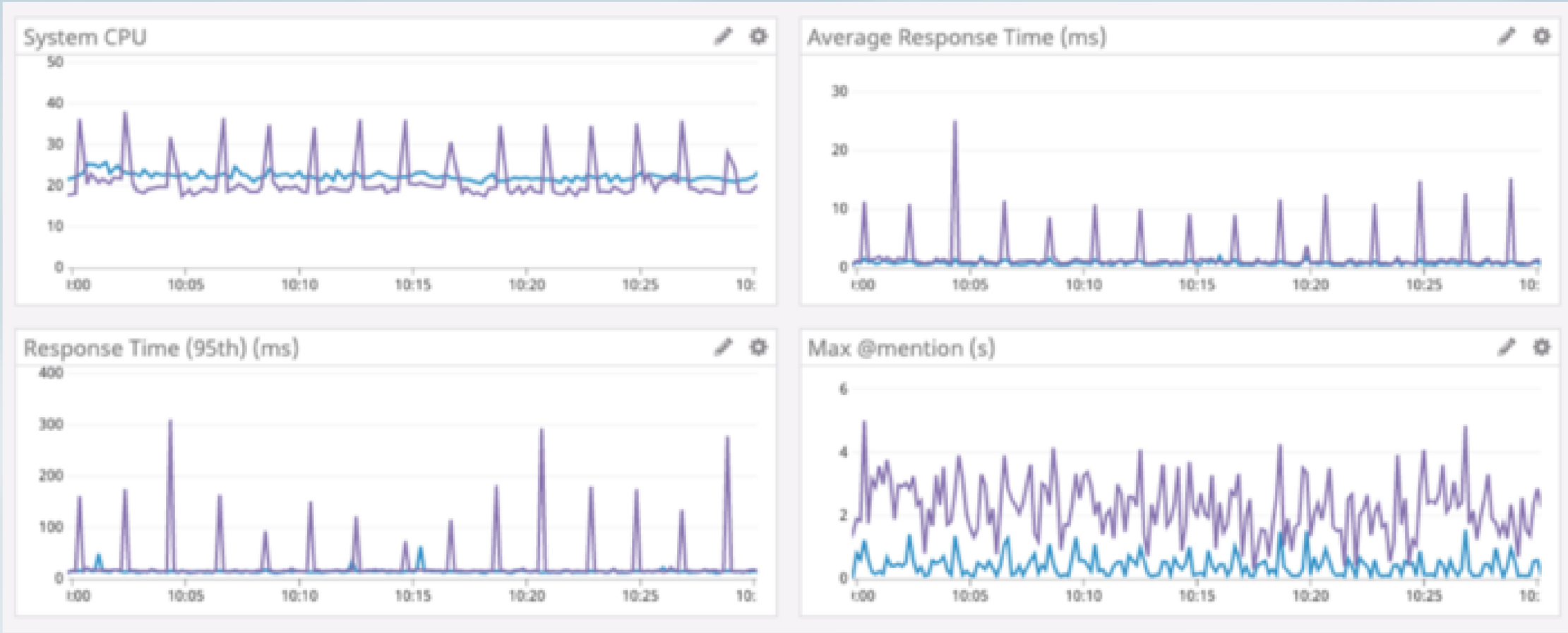
```
fn read_num() -> Result<usize, Box<dyn Error>> {
    // fails with io::Error>
    let s = read_to_string("foo")?;
    // fails with num::ParseIntError>
    let num = s.trim().parse()?;
    Ok(num)
}
```

- **dyn T** means any object implementing *T*
- **Box** wraps an arbitrary type on the heap

OMG!! Is this complicated!

Benefits of this level of control

Discord re-implemented @mention notification in Rust (blue), formerly Go (violet):



Take-aways

Rust's strict programming model shifts projects costs from run time to development time

Is it worth the effort? It depends on the project!

Rust's design takes soundness issues seriously, even if it you have to go an extra mile

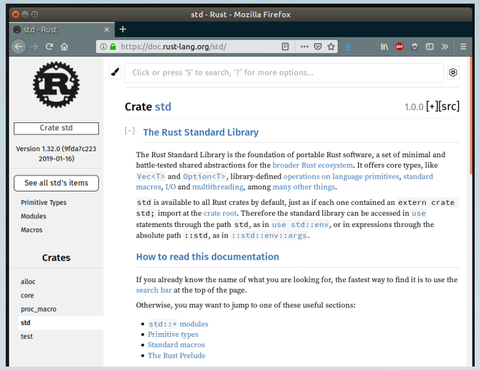
Do I need this level of thoroughness? It depends on the project!

Interactive code example
Embedding Rust code into Python

https://github.com/ckauhaus/mandelbrot-py

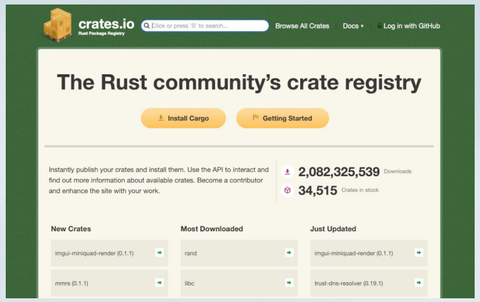
Documentation and further reading

Rust Standard Library



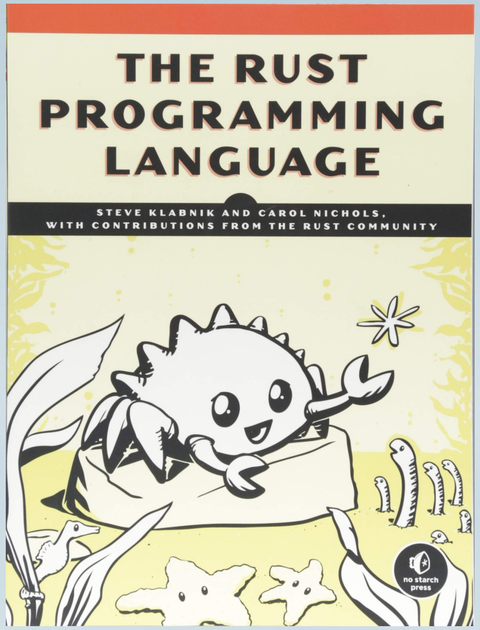
https://doc.rust-lang.org/std/

Crates.io



https://crates.io

The Book



https://doc.rust-lang.org/book/

Crates (libraries)

- Index of Rust libraries: https://lib.rs
- Crate docs: https://docs.rs

Tutorials

- Rust by Example: https://doc.rust-lang.org/rust-by-example/
- Exercism: https://exercism.io/tracks/rust

News

- This Week in Rust: https://this-week-in-rust.org/

Thank you