PyO3: Python Loves Rust

Moshe Zadka – https://cobordism.com

Acknowledgement of Country

Belmont (in San Francisco Bay Area Peninsula)

Ancestral homeland of the Ramaytush Ohlone people

I live in Belmont, in the San Francisco Bay Area Peninsula. I wish to acknowledge it as the ancestral homeland of the Ramaytush Ohlone people.

0.1 Short Intro to Rust

Before talking about how to combine Python with rust, I want to chat a little bit about Rust itself. What is it, why does it exist, and how it looks as a language.

Rust: Intro

What

Why

How

0.1.1 What is Rust?

Rust is a *low-level* language. This means that the things the programmers deals with are *close* to the way computers "really" work.

For example, integer types are defined by bit-size, and correspond to CPU-supported types. While it is tempting to say that this means a+b in Rust corresponds to one machine instructions, it does not mean quite that!

Rust's compiler's chain is non-trivial. It is useful as a first approximation, though, to treat statements like that as "kind of" true.

Rust is designed for *zero-cost abstractions*. This means that a lot of the abstractions available at the *language* level are compiled away at runtime.

For example, unless explicitly asked for, objects are "allocated" on the stack. This means that creating a local object, in Rust, has no runtime cost (though initialization might!)

Finaly, Rust is a *memory-safe* language. There are other memory-safe languages, and there are other zero-cost abstraction languages: usually, those are different languages.

Memory safety does not mean it is *impossible* to have memory violations in Rust. It does mean that there are only two ways that memory violations can happen:

- A bug in the compiler.
- Code which is explicitly declared "unsafe".

Rust standard library code has quite a bit of code which is marked "unsafe", though less than what many assume. This does not make the statement vacuous though: with the (rare) exception of needing to write unsafe code yourself, memory violations will be the results of underlying infrastructure.

Rust: What?

Low-level
Zero-cost abstractions
Memory safe!

0.1.2 Why is Rust?

Why did people create Rust? What problem was not addressed by existing languages?

Rust was designed as a language to achieve a *combination* of high-performance code which is memory safe. This is an increasingly important concern in a *networked* world.

The quintessential use case for Rust is "low-level parsing of protocols". The data which needs to be parsed often comes from untrusted sources, and often needs to be parsed in a performant way.

If this sounds a lot like "what a browser does", this is no coincidence. Rust originally came from the Mozilla foundation, as a way to improve the Firefox browser.

In the modern world, browsers are no longer the only things on which there is both a pressure to be "safe" and "fast". Even the common microservice architecture, combined with defense-in-depth principles, means microservices need to be able to unpack untrusted data "fast".

Rust: Why?

Performance
Safety
"Low-level parsing"

0.1.3 Counting characters

In order to motivate a "wrapping in Rust" example, we need a problem. The problem needs to be

- Easy enough to solve
- Be helped by the ability to write high-performance loops
- Somewhat realistic

The toy problem here is "does a character appear more than X times in a string". This is something that is not easily amenable to performant regular expressions, and even dedicated numpy code can be slower than necessary because often there is no need to scan the entire string.

It is not impossible to imagine some combination of Python libraries and tricks that make this possible. However, the *obvious* algorithm is pretty fast if implemented in a low-level language, and makes things more readable.

In order to make the problem slightly more interesting, and demonstrate some fun parts of Rust, one twist is added. The algorithm supports reseting the count on a newline (does the character appear more than X times in a line) or on a space (does the character appear more than X times in a word).

This is the only nod that will be given to "realism". Any more realism will make the example not useful pedagogically.

Toy Example: Counting

Character appears more than X times
Optionally, reset counts on spaces/newlines
"Toy example"
Just interesting enough

0.1.4 Enum

Rust supports "enums". There are a lot of interesting things that can be done with enums.

For our immediate purposes, a three-way enum without any further twirls is used. The enum encodes "what character resets the count".

Rust example: Enum

```
enum Reset {
    NewlinesReset,
    SpacesReset,
    NoReset,
}
```

0.1.5 Struct

The next Rust thing to be introduced is a bit more substantial: a struct. A Rust struct is somewhat close to a Python "dataclass". Again, there are more sophisticated things that can be done with a struct.

Rust example: Struct

```
struct Counter {
    what: char,
    min_number: u64,
    reset: Reset,
}
```

0.1.6 Implementation

Methods, in Rust, are added to structs in a separate block: the impl block. This has all kinds of reasons which are beyond the current scope.

It does make it convenient to break Rust code into slides! In this example, the method calls an external function. This is mostly done in the service of breaking the code up.

A more sophisticated use would instruct the Rust compiler to inline the function, to allow readability without any runtime cost.

Rust example: Impl

```
impl Counter {
    fn has_count(
        &self,
        data: &str,
    ) -> bool {
        has_count(self, data.chars())
    }
```

0.1.7 Function

Rust variables, by default, are *constant*. Since the current count has to change, it is declared as a *mutable* variable.

The loop goes over the characters, and calls the function got_count. Again, this is done in the service of breaking the code into slides. It does show how to send a *mutable borrow reference* to a function.

Even though current_count is mutable, both the sending site and the receiving site explicitly mark the reference as mutable as well. This makes it clear which functions might modify a value.

Rust example: Loop

```
fn has_count(cntr: &Counter,
      chars: std::str::Chars) -> bool {
    let mut current_count : u64 = 0;
    for c in chars {
        if got_count(cntr, c, &mut current_count) {
            return true;
        }
    }
    false
}
```

0.1.8 Counting

The got_count resets the counter, increments the counter, and then checks it. Rust colon-separated sequence of expressions evaluates to the result of the last expression, in this case, whether the threshold was met.

Rust example: Counting

```
fn got_count(cntr: &Counter,
      c: char, current_count: &mut u64) -> bool {
    maybe_reset(cntr, c, current_count);
    maybe_incr(cntr, c, current_count);
    *current_count >= cntr.min_number
}
```

0.1.9 Reset

The reset code shows another useful thing in Rust: matching. A complete description of the matching abilities in Rust would be a semester-level class, not two minutes in an unrelated talk, but here we match on a tuple matching one of two options.

Rust example: Reset

```
fn maybe_reset(cntr: &Counter,
      c: char, current_count: &mut u64) -> () {
    match (c, cntr.reset) {
        ('\n', Reset::NewlinesReset) |
        (' ', Reset::SpacesReset)=> {
            *current_count = 0;
        }
        _ => {}
    };
}
```

0.1.10 Increment

The increment compares the character to the desired one, and if so, increments the count.

Rust example: Increment

```
fn maybe_incr(cntr: &Counter,
      c: char, current_count: &mut u64) -> (){
    if c == cntr.what {
        *current_count += 1;
    };
}
```

0.1.11 Wrap-up

Note that the code here was optimized for slides. It is not necessarily a best-practice example of Rust code or how to design a good API.

Rust example: disclaimer

Not necessarily best practices:
Code style
API

0.2 PyO3

In order to wrap the code for Python, we use PyO3. The PyO3 Rust "crate" (or library) allows *inlining* the hints for how to wrap Rust code into Python. This makes it easier to modify both together.

PyO3

Inline
Modify together

0.2.1 Include

The first step is to *include* the pyo3 crate primitives.

PyO3 example: Include

```
use pyo3::prelude::*;
```

0.2.2 Wrap enum

The enum needs to be wrapped. The derive clauses are necessary for wrapping the enum for PyO3, since they make the class copyable and clonable, which makes them easier to use from Python.

PyO3 example: Wrap enum

```
#[pyclass]
#[derive(Clone)]
#[derive(Copy)]
enum Reset {
    /* ... */
}
```

0.2.3 Wrap struct

Similarly, the struct is wrapped in the same way. These call "macros" in Rust which generate the needed interface bits.

PyO3 example: Wrap struct

```
#[pyclass]
struct Counter {
    /* ... */
}
```

0.2.4 Wrap impl

Wrapping the impl is more interesting. A new method, new is added. This method is marked as #[new], which lets PyO3 know how to expose a constructor for the built-in object.

PyO3 example: Wrap impl

```
#[pymethods]
impl Counter {
    #[new]
    fn new(what: char, min_number: u64,
           reset: Reset) -> Self {
        Counter{what: what,
          min_number: min_number, reset: reset}
    }
    /* ... */
}
```

0.2.5 Define module

Finally, a function that initializes the module needs to be defined. This function has a specific signature, must be named the same as the module, and decorated with #[pymodule].

The ? show that this function can *fail* (for example, if the class was not appropriately configured). The PyResult is translated into a Python exception at import time.

PyO3 example: Define module

```
#[pymodule]
fn counter(_py: Python, m: &PyModule
) -> PyResult<()> {
    m.add_class::<Counter>()?;
    m.add_class::<Reset>()?;
    Ok(())
}
```

0.2.6 Maturin develop

For quick checking, maturin develop will build and install the library into the current virtual environment. This helps iterate quickly.

Maturin develop

```
(venv)$ maturin develop
```

0.2.7 Maturin build

The maturin build command builds a manylinux wheel, which can be uploaded to PyPI. Remember that the wheel is specific to the CPU architecture.

Maturin build

```
(venv)$ maturin build
```

0.3 Python

Using the library from Python is the nice part. There is nothing to indicate a difference between this and writing the code in Python.

One useful aspect of this is that if you are optimizing an existing library in Python, which already has unit tests, you can use the Python unit tests as the tests for the Rust library.

Python
Use!

0.3.1 Import

Whether this is because maturin develop installed it or pip install installed a wheel, importing the library is done with import.

Import

```
import counter
```

0.3.2 Construct

The constructor was defined exactly so that the object could be built from Python. Note that this is not always the case: sometimes objects are only returned from more sophisticated functions.

Constructor

```
cntr = counter.Counter(
    'c',
    3,
    counter.Reset.NewlinesReset,
)
```

0.3.3 Call

The final pay-off is now here: check whether this string has at least three "c" characters:

Call

```
cntr.has_count("hello-c-c-c-goodbye")
```

```
True
```

0.3.4 Call

Adding a newline causes the rest to happen, and there will not be three "c" characters without an intervening newline:

Call

```
cntr.has_count("hello-c-c-\nc-goodbye")
```

```
False
```

0.4 Conclusion

Why is this talk useful? Why am I speaking about this?

Take-aways
Why?

0.4.1 Rust and Python is easy

My goal is to convince you that combining Rust and Python is easy. I wrote little code to "glue" them.

Rust + Python
Easy!

0.4.2 Use each one for its purposes

Rust and Python have complementary strengths and weaknesses.

Rust is great for high-performance, safe code. Rust has a steep learning curve and can be awkward for quickly prototyping a solution.

Python is easy to get started with, and supports incredibly tight iteration loops. Python does have a "speed cap": beyond a certain level, it is harder to get better performance from Python.

Differences

Rust: High-performance, safe, learning curve, awkward prototyping

Python: Easy, tight iteration, Speed cap

Combining them is perfect. Prototype in Python, and move performance bottlenecks to Rust.

Combined

Prototype in Python

Move perf bottlenecks to Rust

With maturin, your development and deployment pipelines are easier to make. Develop, build, and enjoy the combo!

Stronger together

Development

Deployment

Enjoy!