Owen Synge

Why would a python developer learn rust, when there are no jobs in it?

HEPiX, October 2018

General opinions on programming languages

- Must interface with C well. (Or maybe Java ….)
- Must have a very significant advantage
  - For example, python is:
    - Very fast to develop / Very easy to test ….
- Must be able to do all the common tasks I need.
  - So must have good libraries for:
    - Testing, CLI, RDBMS, Logging, JSON, etc …
  - I test all of these use cases before I can recommend.
- Rust passes my evaluation with flying colours!
  - Otherwise I would not give this talk.
  - Very different advantages than python.

Why I like Python

- Fast to write.
- Quick to debug.
- Fast enough for most use cases.
  - Approximately 100x slower than C
- Duck typing makes it succinct.
- Good enough error handling.
- I like “Python OO” better than “C++ OO”.
- Tooling/Packaging is quick.
  - Tox, virtual env, pytest, and mock make testing a pleasure.
  - Sadly had to learn much python magic in setup.py
  - Making rpm / debian package is easy.

With python you have compromises

- Multithreading
  - GIL (Global Interpretor Lock) is sometimes a blocking issue.
- Performance
- Embedded
  - OpenWRT
- Memory Usage
  - On router or phone.
- Python SWIG issues between 2 and 3.
  - M2crypto only just started working with python 3 (Hmm.. maybe I was wrong)
- Bugs in large projects
  - Sometimes python just ignores bad code and uses an alternative path!
    - I have not seen this recently but it was true in past.

Working around Python limitations.

- Revolutionary:
  - Rewrite Python code “prototype/production” as C/C++/D/ADA/Go/Rust ...
- Incremental:
  - Python extensions.
    - Native C binding
      - https://docs.python.org/3/extending/building.html
    - Swig can automate wrapping of C methods. (auto-generated code dangers)
    - Ideally integrated in setup.py
      - C++ with PyCXX
      - Rust with PyO3
      - Go with go-pymodule
  - Pythons FFI can call C methods.
    - Allows C/C++

What makes me avoid C?

- Development is very slow.
- Debugging is hard.
- Multithreading is hard.
- Compiler errors.
- Memory management issues.
  - Often only appear in production.

What makes me avoid C++?

- No stable ABI
  - Have to do work with C interfaces.
- Development is slow.
- Debugging is hard.
- Multithreading is hard.
- Compiler errors.
- Memory management issues.
  - Although this is easier with modern C++ eg C++17

Why did I Try Rust?

- I heard great things about rust. (CCC and FOSDEM)
  - Borrow checking?
- Native and compiled use cases.
- Extending python.
- To be as fast as C/C++ when needed.
  - C++ like philosophy
    - Abstraction without overhead.
  - No Garbage Collection due to borrow checking?
    - Language can be realtime.
    - No runtime environment
- Maybe rust helps with multithreading?
- Maybe 10 years after python something new came out?
  - Rust was first announced approximately ten years after python.

Rust: Interesting design decisions

- Designed to replace C/C++ in Firefox.
  - Designed for incremental adoption in C/C++ code.
- Zero abstraction overhead philosophy just like C++.
- No garbage collector → but has a borrow checker!
- Multithreaded support baked into type system.
  - Compiling guarantees thread safety.
    - As example ‘Mutex locking’ is implemented as a generic type: std::sync::Mutex<T>
      - So you have to use the mutex lock to get access to data, and unlocks automatically as leaves scope.
- Variables are immutable by default.
  - Sadly we can not add this to C/C++ now.
- Enumerated types can contain variables.
  - Common in modern languages → Implications are surprising and good.
    - The ML family of languages had this back in the day.
- Keep the language small, and flexible.
  - The standard rust libraries are in Rust and optional.

My first experiences of Rust

- Lots in rust is unfamiliar to me.
  - Rustup (Tool chain updater)
  - Cargo (Dependency management)
  - Structures +Traits Vs Objects (Why reinvent wheel?)
  - Error handling (No exceptions!)
  - Compiler (LLVM rather than gcc)
  - The borrow checker (This seems revolutionary!)
- Maturity questions worried me.
  - Rust stable, tool chain etc
  - Rust crates, are their enough doing useful things.

Rust: Rustup

- The “default” way to get rust development tools.
- Install downloads and executes stuff from web.
  - Not very happy about this.
  - Can see advantages for a language development.
  - Can see advantages for a language not yet packaged.
- Can be avoided for rust stable.
  - Distros are now packaging some of the rust tools.
    - Rust and cargo now packaged for SL7, Fedora, SUSE, and debian testing
- If you use rust nightly or cross compilation you will want rustup.
  - You may still need rust nightly (see later in talk)

Rust: Cargo

- Downloads and builds dependencies.
  - Expects internet connection
  - With nightly we have cargo-vendor
    - So can build off line / Repeatable builds.
    - Update: has now reached stable :)
- Simple to use.
  - But too simple for some use cases.
- Can be called by auto-tools (not tried with cmake)
  - When you need some thing cargo can’t do natively.
    - Eg make template files, check C libraries and headers are installed.
- autogenerate code with cargo build process.
  - Build.rs file compiles and runs before rest of cargo process.
  - Bindgen which wraps C is a great example (Many others exist)

Rust: Compiler

- Error messages are mostly very helpful.
  - Many times even propose correct solution to issue.
- Macros show errors before being expanded.
  - Unlike C++ can find your mistake fast
    - Do not get pages of compile errors.
- Slow compile speeds.
  - Slightly worse than very template heavy C++.
  - No where need the speed of golang compiler
- Borrow checker is lovely
  - But compiler does feel like it is mean at first.

Rust: Error Handling

- Rust does not have exceptions like C++/python.
- Error handling is encouraged by language.
  - Not forced but ..
- Unrecoverable errors like assert in C.
  - !Panic → Gives stack trace if compiled with debug flag.
- Recoverable errors like like C but better.
  - Makes use of enumerated types.
    - Result<T,E> → Great abstraction for normal path errors.
    - Rather like C but safer.
- Error handling does seems verbose.
  - Macros, “?”, and libraries like ErrorChain remove lots of boilerplate

Rust: is not quiet OO.

- Seem to provide what I want from Object Orientated.
  - But does not support inheritance.
  - Use the “Has a” not “Is a” model.
- Structures can have methods.
  - Methods can be added outside the library.
- Traits allow methods for more than one structure.
  - Seems clearer in code than Polymorphic Objects.
  - Traits can be added outside the library.
- Have yet to see down side.

Rust examples of OO like features.

Binding method to structure

```
1  struct Point {
2      x: f64,
3      y: f64,
4  }
5
6  // a free-standing function that converts a (borrowed) point to a string
7  fn point_to_string(point: &Point) -> String { ... }
8
9  // an "inherent impl" block defines the methods available directly on a type
10 impl Point {
11     // this method is available on any Point, and automatically borrows the
12     // Point value
13     fn to_string(&self) -> String { ... }
14 }
```

Rust examples of OO like features.

Traits: A common method to multiple types

```
 1  trait Hash {
 2      fn hash(&self) -> u64;
 3  }
 4
 5
 6  impl Hash for bool {
 7      fn hash(&self) -> u64 {
 8          if *self { 0 } else { 1 }
 9      }
10  }
11
12  impl Hash for i64 {
13      fn hash(&self) -> u64 {
14          *self as u64
15      }
16  }
```

Rust examples of OO like features.

Using serde macros on structs

```rust
#[macro_use]
extern crate serde_derive;

extern crate serde;
extern crate serde_json;

#[derive(Serialize, Deserialize, Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 1, y: 2 };

    // Convert the Point to a JSON string.
    let serialized = serde_json::to_string(&point).unwrap();

    // Prints serialized = {"x":1,"y":2}
    println!("serialized = {}", serialized);

    // Convert the JSON string back to a Point.
    let deserialized: Point = serde_json::from_str(&serialized).unwrap();

    // Prints deserialized = Point { x: 1, y: 2 }
    println!("deserialized = {:?}", deserialized);
}
```

Rust: Borrow checker

- Ownership of variables/memory is part of the type system.
  - Only one mutable owner of data at a time.
  - Functions can borrow ownership of variables/memory
- Manages lifetime of variables/memory.
  - Prevents references after variables/memory is moved or dropped.
    - More general solution than reference counting.
- Like “garbage collection at compile time.”
  - Reduces development time and bugs.
  - Allows for real time code.
- Thread safe checking at compile time
- Does not need a run time garbage collection.
  - Rust executables are larger than C but smaller than Go.
- Think of it like C++ static analysis built into the language with advantages.
  - To provide memory management.
  - To provide thread safety.
- Apple’s new “swift” language will get a borrow checker in next version too!

Using pyo3 to embed rust in python

- Pros:
  - Very fast execution
  - Allows multithreading
  - Very easy
  - Get rust safety
  - Macros do all hard work
  - Use setup.py to make it transparent
- Cons:
  - Only works with rust nightly
  - Need to install library to make setup.py work
  - API had changed with latest release (but it is for the better...)

Embedding rust in python using pyo3

Rust code called by python

```rust
// Source adapted from
// https://github.com/tildeio/helix-website/blob/master/crates/word_count/src/lib.rs
#![feature(specialization)]

#[macro_use]
extern crate pyo3;
extern crate rayon;

use pyo3::prelude::*;
use rayon::prelude::*;
use std::fs;
use std::path::PathBuf;

/// Represents a file that can be searched
#[pyclass]
struct WordCounter {
    path: PathBuf,
}

#[pymethods]
impl WordCounter {
    #[new]
    fn __new__(obj: &PyRawObject, path: String) -> PyResult<()> {
        obj.init(|_| WordCounter {
            path: PathBuf::from(path),
        })
    }

    /// Searches for the word, parallelized by rayon
    fn search(&self, py: Python, search: String) -> PyResult<usize> {
        let contents = fs::read_to_string(&self.path)?;

        let count = py.allow_threads(move || {
            contents
                .par_lines()
                .map(|line| count_line(line, &search))
                .sum()
        });
        Ok(count)
    }

    /// Searches for a word in a classic sequential fashion
    fn search_sequential(&self, needle: String) -> PyResult<usize> {
        let contents = fs::read_to_string(&self.path)?;

        let result = contents.lines().map(|line| count_line(line, &needle)).sum();

        Ok(result)
    }
}

fn matches(word: &str, needle: &str) -> bool {
    let mut needle = needle.chars();
    for ch in word.chars().skip_while(|ch| !ch.is_alphabetic()) {
        match needle.next() {
            None => {
                return !ch.is_alphabetic();
            }
            Some(expect) => {
                if ch.to_lowercase().next() != Some(expect) {
                    return false;
                }
            }
        }
    }
    return needle.next().is_none();
}

/// Count the occurences of needle in line, case insensitive
#[pyfunction]
fn count_line(line: &str, needle: &str) -> usize {
    let mut total = 0;
    for word in line.split(' ') {
        if matches(word, needle) {
            total += 1;
        }
    }
    total
}

#[pymodinit]
fn word_count(_py: Python, m: &PyModule) -> PyResult<()> {
    m.add_function(wrap_function!(count_line))?;
    m.add_class::<WordCounter>()?;

    Ok(())
}
```

Python tests calling rust methods

```python
# -*- coding: utf-8 -*-
import os

import pytest
import word_count

current_dir = os.path.abspath(os.path.dirname(__file__))
path = os.path.join(current_dir, "zen-of-python.txt")


@pytest.fixture(scope="session", autouse=True)
def textfile():
    text = """
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
"""

    with open(path, "w") as f:
        f.write(text * 1000)
    yield
    os.remove(path)


def test_word_count_rust_parallel(benchmark):
    count = benchmark(word_count.WordCounter(path).search, "is")
    assert count == 10000


def test_word_count_rust_sequential(benchmark):
    count = benchmark(word_count.WordCounter(path).search_sequential, "is")
    assert count == 10000


def test_word_count_python_sequential(benchmark):
    count = benchmark(word_count.search_py, path, "is")
    assert count == 10000
```

__init__.py for rust module

```python
# -*- coding: utf-8 -*-
from .word_count import WordCounter, count_line

__all__ = ["WordCounter", "count_line", "search_py"]


def search_py(path, needle):
    total = 0
    with open(path, "r") as f:
        for line in f:
            words = line.split(" ")
            for word in words:
                if word == needle:
                    total += 1
    return total

```

File list of example python module in rust

```
usr/
usr/local
usr/local/lib
usr/local/lib/python2.7
usr/local/lib/python2.7/dist-packages
usr/local/lib/python2.7/dist-packages/word_count
usr/local/lib/python2.7/dist-packages/word_count/__init__.py
usr/local/lib/python2.7/dist-packages/word_count/word_count.so
usr/local/lib/python2.7/dist-packages/word_count/__init__.pyc
usr/local/lib/python2.7/dist-packages/word_count-0.1.0.egg-info
usr/local/lib/python2.7/dist-packages/word_count-0.1.0.egg-info/PKG-INFO
usr/local/lib/python2.7/dist-packages/word_count-0.1.0.egg-info/not-zip-safe
usr/local/lib/python2.7/dist-packages/word_count-0.1.0.egg-info/top_level.txt
usr/local/lib/python2.7/dist-packages/word_count-0.1.0.egg-info/dependency_links.txt
usr/local/lib/python2.7/dist-packages/word_count-0.1.0.egg-info/SOURCES.txt
$
```

Examples from pyo3 git repository: https://github.com/PyO3/pyo3/tree/master/examples/word-count

Rust: Issues I have found

- Only implemented on LLVM compiler.
  - Not 100% self supporting.
- Must use C-ABI, no compiler neutral ABI
  - Just like C++.
  - Very little work to make Rust export a C ABI
    - Just use some macros on functions and structures with C compatible types.
- Rust is not easy to learn.
  - While simpler than C++ it is much harder to get started with than python.
  - Mostly due to not allowing you to compile code that could have undefined behaviour.
  - Some design patterns have to be adjusted.
- By default cargo downloads dependencies from internet.
  - Hence repeatable builds more complex than C/C++.
  - Cargo has vendorise in rust nightly.
    - Update → In stable now for a few months.

Rust: Issues I have found

- Async IO story is only just maturing.
  - Tokio, Futures, generators (Libraries From stable, new, and experimental)
- Some things require Rust nightly (Was Many)
  - Incremental builds only just reached stable.
    - To speed compile time
  - Meta-programing tools. (ie code about code)
    - PyO3, mocking libraries, etc
    - I can see progress here
- Writing “unsafe” rust (when you cant use rust safety eg linking to C)
  - When using “unsafe” keyword rust compiler does not help me over C.
  - Fortunately not often needed.

Rust: Issues for a Python dev.

- Because I am coming from python
  - Rust syntax: treatment of ‘;’ is annoying
    - Line without ‘;’ is function return value.
      - If function returns no values will terminate function.
      - Can produce misleading type errors in compiler output.
  - Mocking in python is so nice
    - Have to mock more code in static compiled languages.
  - ipython (interactive python) is very helpful
    - Unmatched as a quick way to explore a libraries/API.
  - Rust is more verbose than python.

So why do I like rust?

- Fast to write code (not as fast as python).
- Easy to embed in python with pyo3 (https://github.com/PyO3/pyo3)
  - Needs to use rust nightly
  - Maybe too bleading edge for production.
- Compiling code nearly always does what you expect first time. (Unlike Python/C)
- Executes very fast, uses little memory, and produces smallish libraries and executables.
- Has a great set of libraries that allow me to get things done.
  - Many inspired by C++ and python.
- Designed for incremental adoption.
  - Look to “libsvg” for great blogs and talks on this.

So why do I like rust?

- Designed for incremental adoption.
  - Look to “libsvg” for great blogs and talks on this.
- Rust is a nearly “universal language”.
  - Can replace C/C++ for most use cases.
    - If you can use LLVM compiler, I believe you can use rust rather than C/C++.
- Consistent and Succinct language
  - Type inference and many other tricks.
  - Type system is great.
  - Generics are well supported.
  - No null exceptions
- Borrow checker is a great compromise.
  - Makes me feel safe in my code.
  - Prefer this to a garbage collector for a low level language.
  - Eases binding to other languages. eg C/python …
  - Makes you concerned when writing C/C++

Why would a python developer learn rust, when there are no jobs in it?

- Rust is growing fast.
  - Still young (1.0 release was May 15, 2015)
  - Some jobs adverts now exist. (was not true a year ago)
  - Some metric put it in top 20 Languages (most in top 30)
- The “Most Loved” programming language.
  - For the third year in a row, Stack overflow survey.
- A great way to extend python/C/C++ projects.
- While C++ 17 seems a great improvement
  - I prefer what rust has to offer.
- Rust has yet to disappoint me.

References

- Rust home page
  - https://www.rust-lang.org/en-US/
- Rust libraries
  - https://crates.io/
- Binding C from rust
  - https://rust-lang-nursery.github.io/rust-bindgen/introduction.html
- Binding rust from python
  - https://github.com/PyO3/pyo3
- Great tour of rust features
  - http://zsiciarz.github.io/24daysofrust/
- Libsvg and its migration from C/C++ to rust
  - https://people.gnome.org/~federico/blog/librsvg-posts.html
- Using ECS to do simulations for game dev in rust
  - https://www.youtube.com/watch?v=aKLntZcp27M
- Companies using rust in production
  - https://www.rust-lang.org/en-US/friends.html

Some big companies using rust in production.

Rewrote OSD's, erasure coding, and Bulk data transfers in rust from Golang.

Use Rust in a service for analysing petabytes of source code.

Replacing C and rewriting performance-critical bottlenecks in the registry service architecture.

Replacing memory-unsafe languages (particularly C) and are using it in the core edge logic.

Questions

- Expected questions:
  - Is rust mature enough for HEP(iX) to use?
  - Would you recommend HEP(iX) uses rust?
  - Will rust replace python/C/C++/Golang?
  - When would I use rust?
  - How would I compare rust with C/C++/Golang?
    - Performance, memory usage, speed, stability?
    - Maturity, toolchain, libraries, magic?
  - When do you need to use “unsafe” in rust?
  - Any lessons learnt from Multithreading in rust?