I consider myself mainly a Python developer these days, but coming from a history of C++ development, I've been really interested in seeing how Rust pans out. Could it be an ideal language to use when I need the performance that Python just can't provide? Can I happily replace C or C++ (using boost) with Rust?

I set out to see the current state of binding Python to Rust.

Python

Let's begin with a fibonacci sequence calculator. It's pretty basic in Python:

```python
def fib(n):
  if n < 2:
    return 1
  prev1 = 1
  prev2 = 1
  for i in range(1, n):
    next = prev1 + prev2
    prev2 = prev1
    prev1 = next

  return prev1
```

python_example.py hosted with ♥ by GitHub view raw

C

C has a lot of boiler plate where I could just use the `cffi` module. But using the actual API is worth it in case you want to do any sort of error handling in the C side. For example, if you pass a wrong type through `cffi`, you basically just segfault. You can wrap your C code with error handling in Python. But if the intent is to make a function you call in a loop, you want the error checking to also be fast.

If I'm honest, I always feel like I'm putting in a lot of bugs when I use the C API. Especially when I compare it with the Cython output.

```c
#include <Python.h>
#include <stdint.h>

static uint64_t _fib(uint64_t n) {
    if (n == 1 || n == 2) {
        return 1;
    }
    uint64_t prev1 = 1;
    uint64_t prev2 = 1;
    for (uint64_t i = 1; i < n; ++i) {
        uint64_t next = prev1 + prev2;
```

```
        prev2 = prev1;
        prev1 = next;
    }
    return prev1;
}

static PyObject* fib(PyObject* self, PyObject* args) {
    int64_t n;

    if (!PyArg_ParseTuple(args, "L", &n)) {
        return NULL;
    }
    if (n < 0) {
        return NULL;
    }
    uint64_t f = _fib(n);
    return Py_BuildValue("L", f);
}

static PyMethodDef c_python_methods[] =
{
     {"fib", fib, METH_VARARGS, "Compute a fibonacci sequence value."},
     {NULL, NULL, 0, NULL}
};

static struct PyModuleDef c_python_module = {
    PyModuleDef_HEAD_INIT,
    "c_python_exaplme",
    "Example module",
    -1,
    c_python_methods,
    NULL,
    NULL,
    NULL,
    NULL
};

PyMODINIT_FUNC
PyInit_c_python_example(void)
{
     return PyModule_Create(&c_python_module);
}
```

c_python_example.c hosted with ♥ by **GitHub** **view raw**

```
.PHONY: all

CFLAGS=$(shell pkg-config --libs --cflags python3) -O3 -Wall -Werror -pedantic -march=native
all: c_python_example.so
```

```
5
6  c_python_example.so: c_python_example.c
7          $(CC) $(CFLAGS) $< -o $@
8
9  clean:
10         rm c_python_example.so
```

Makefile hosted with ♥ by **GitHub** view raw

C++ (Boost.Python)

Here's what the equivalent looks like in Boost.Python. Boost.Python is a really nice library for wrapping C++ to call from Python. Except if you manage to make a mistake you get all the associated C++ compilation errors some of us love to hate. It's allegedly gotten better, but in making this example, I still found some error explosions which take some searching around on the internet to figure out.

Sadly Boost.Python only seems to support Python 2. I know a lot of people using Python 2 who have no interest in migrating to Python 3, but if you end up wrapping a lot of your code in Boost.Python, you're really sealing your fate. I don't recommend it.

```
1  #include <boost/python.hpp>
2
3  static uint64_t fib(uint64_t n) {
4      if (n == 1 || n == 2) {
5          return 1;
6      }
7
8      uint64_t prev1 = 1;
9      uint64_t prev2 = 1;
10     for (uint64_t i = 1; i < n; ++i) {
11         uint64_t next = prev1 + prev2;
12         prev2 = prev1;
13         prev1 = next;
14     }
15     return prev1;
16 }
17
18 BOOST_PYTHON_MODULE(cc_python_example) {
19     using namespace boost::python;
20     def("fib", fib);
21 }
```

cc_python_example.cc hosted with ♥ by **GitHub** view raw

```
1  .PHONY: all
2
3  CFLAGS=$(shell pkg-config --libs --cflags python) -lboost_python -O3 -Wall -Werror -pedantic
```

```
all: cc_python_example.so

cc_python_example.so: cc_python_example.cc
	$(CXX) $(CFLAGS) $< -o $@

clean:
	rm cc_python_example.so
```

Makefile hosted with ♥ by **GitHub** view raw

Cython

Cython is a fantastic tool for writing Python bindings. It's a dialect of Python that looks so much like Python that you can almost copy paste your code into a `pyx` module, configure `setuptools` to find it, and you're running at a much faster speed. If you take a look at the Cython code, set the types on the variables, turn off array bounds checking, and other tweaks, it competes with hand written C. It's really awesome.

```
def fib(int n):
    if n < 2:
        return 1
    cdef unsigned long long prev1 = 1
    cdef unsigned long long prev2 = 1
    cdef int i = 1
    while i < n:
        next = prev1 + prev2
        prev2 = prev1
        prev1 = next
        i += 1
    return prev1
```

cython_python_example.pyx hosted with ♥ by **GitHub** view raw

Rust (rust-cpython)

The best library I've found for writing modules in Rust is rust-cpython. Using it took some trial and error since it's so new and I wouldn't call myself a Rustacean just yet. But it was actually a very pleasant experience. With a little manipulation of the `Cargo.toml` file (used to define a reproducable build in Rust), you can build against Python 2 or 3. None of the other wrappers, aside from Cython can do this.

Finally, here's how the Fibonacci number generator looks in Rust:

```
[package]
name = "rust_python_example"
version = "0.1.0"
authors = []
```

```toml

[lib]
name = "rust_python_example"
crate-type = ["dylib"]

[dependencies]
interpolate_idents = "*"

[dependencies.cpython]
version = "*"
default-features = false
features = ["python3-sys"]
```

Cargo.toml hosted with ♥ by **GitHub** **view raw**

```rust
#![feature(plugin)]
#![plugin(interpolate_idents)]

#[macro_use] extern crate cpython;

use cpython::{PyResult, Python, PyTuple, PyErr, exc, ToPyObject, PythonObject};

mod fib {

pub fn fib(n : u64) -> u64 {
    if n < 2 {
        return 1
    }
    let mut prev1 = 1;
    let mut prev2 = 1;
    for _ in 1..n {
        let new = prev1 + prev2;
        prev2 = prev1;
        prev1 = new;
    }
    prev1
}
}

py_module_initializer!(librust_python_example, |_py, m| {
    try!(m.add("__doc__", "Module documentation string"));
    try!(m.add("fib", py_fn!(fib)));
    Ok(())
});

fn fib<'p>(py: Python<'p>, args: &PyTuple<'p>) -> PyResult<'p, u64> {
    let arg0 = match args.get_item(0).extract::<u64>() {
        Ok(x) => x,
        Err(_) => {
```

```
            let msg = "Fib takes a number greater than 0";
            let pyerr = PyErr::new_lazy_init(py.get_type::<exc::ValueError>(), Some(msg.to_py
            return Err(pyerr);
        }
    };
    Ok(fib::fib(arg0))
}
```

lib.rs hosted with ♥ by GitHub view raw

I ran some basic tests so see how the performance fares, and it turns out that it's pretty competetive. I wouldn't put too much stock into the actual numbers aside from the fact that they're within an order of magnitude of each other. I used GCC for C and Rust is built on LLVM.

The test is a really noddy one. It's just using 'timeit' on the 91st fibonacci number from the IPython shell:

```
timeit fib(90)
```

Python Version	Implementing language	timeit result
Python 3.4.2	Python	100000 loops, best of 3: 7.05 µs per loop
Python 3.4.2	C	1000000 loops, best of 3: 288 ns per loop
Python 3.4.2	Rust	1000000 loops, best of 3: 229 ns per loop
Python 3.4.2	Cython	10000000 loops, best of 3: 192 ns per loop
Python 2.7.10	Python	100000 loops, best of 3: 6.11 µs per loop
Python 2.7.10	C++	1000000 loops, best of 3: 219 ns per loop
Python 2.7.10	Rust	1000000 loops, best of 3: 177 ns per loop
Python 2.7.10	Cython	10000000 loops, best of 3: 171 ns per loop

As it's a microbenchmark, the only thing I would say with any amount of confidence is that the Rust API wrapper probably isn't getting in the way too much.

Current work

Some Rustaceans/Pythonistas are working on this infrastructure and it's pretty exciting. Other pieces of work being put into place are integration with Python's setuptools so you can run `python setup.py install` and it will be able to build the Python modules just like you would expect.

Current issues

1. I haven't found anyone who is looking to support Numpy yet, so anyone who wants to do their numeric/scientific development in Python/Rust should hold off at the moment.
2. I don't see how to specify the library name in Cargo. This means all the shared objects are always prefixed with `lib`. This means you need to import the module as `import libXYZ` which isn't really what you want.

Further work

I haven't tested managing objects on the Rust side, but this is apparently a place where Python can potentially get some performance improvements. For example, collections.OrderedDict was implemented in raw Python until Python 3.5. Of course, the ability to use RAII on the Rust side should make writing containers in Rust a lot easier than in C.

Conclusion

Rust looks like a really great language to implement Python modules. It's not there yet if you want to integrate with the numeric/stats/machine learning libraries that Python offers. But with some love and more community effort, I could see a community building in this direction. I think it's a very promising start and I hope it continues.

Related Posts

Ewan Higgs

github.com/ehiggs
bitbucket.org/ewanhiggs
be.linkedin.com/in/ewanhiggs