Rust and its usage as Python extensions

PyGamma 2019 Heidelberg

Matthieu Baumann

03/19/19

Summary

1. Rust programming language introduction
2. Use of Rust extension codes into the **cdshealpix** Python package
3. **cdshealpix** deployment for Windows, MacOS and Linux

Part I: Rust programming language presentation

Rust Presentation

- Rust is a compiled system programming language (no garbage collector!)
- It tries to detect as much errors as possible statically (i.e. during the compilation)
- Therefore, it embeds some "rules" to guide/force you to code in a safety way
- These rules prevent your code to have segmentation faults, dereference null pointers, etc...

What are these “rules” about ?

The ownership concept

- At any time, a resource is owned by exactly one scope!
- When the resource goes out of its scope, it gets freed

The borrowing

- A scope (e.g. other methods) can borrow a resource: this is done by references
- Two types of borrowing: immutably (&, default behaviour) and mutably (&mut)
- When the reference goes out of the scope, the ownership is restored to the caller. The resource is not dropped
- At any time, you can either have:
  - one and only one mut ref to a resource
  - several immutable refs to the same resource

Lifetime annotation of references

- lifetime annotations ensure that referenced resources always outlive object instances that refer them.

Some Rust nice features

- The cargo package manager. All rust dependency libs (called **crates**) are written in a *Cargo.toml* configuration file at the root of the project.

```
[package]
name = "cdshealpix_python"
version = "0.1.10"
...

[dependencies]
# From github repo
cdshealpix = { git = 'https://github.com/
 cds-astro/cds-healpix-rust', branch = 'master' }
# or from crates.io
cdshealpix = "0.1.5"
```

- Safety: ownership, borrowing, lifetimes
- Performance:
  - No garbage collector but strong rules checked during the compilation! This force the programmer to code in a "safer" way, think about the reference lifetimes etc...
  - Zero-cost abstractions:
    - *Common collections* given by the standard library: Vec, HashMap
    - *Generics*: statically generation of Rust code auto-inlined by the compiler.
    - Iterators with *map*, *filter*, ..., defined on them
    - Lambda functions (called *closures*)
    - *Object oriented*, Traits are java-like interfaces, no data attribute inheritance.
    - *Error handling*
    - Strong typing and type inference
- *Concurrency*: some primitives implemented in the std library: Mutexes, RWLocks, Atomics.
- See the well-explained *official documentation* and *Rust by examples* for more infos!

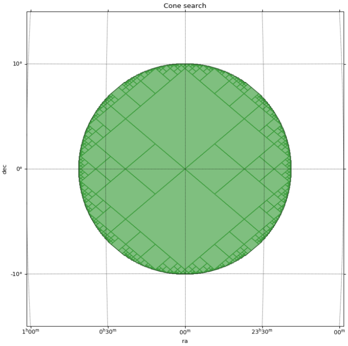
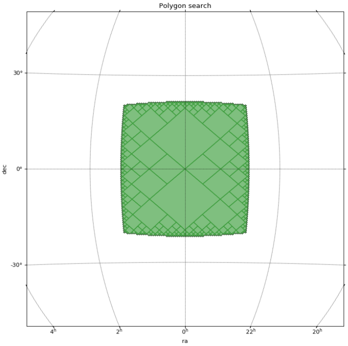
Where is Rust used and by who ?

- Quite new: 1.0.0 released in 2015
- *Most Loved languages*. Rust is 1st, Kotlin 2nd, Python 3rd, ..., C++ 22th. For the third year in a row **Rust** is the **most loved** language.
- Begin to be used in the game industry as a replacement for C++. See *here*.
- Over 70% of developers who work with Rust contribute to **open source** (*stackoverflow latest 2018 survey*)

Part II: use of Rust extension codes into the **cdshealpix** Python package

cdshealpix presentation

- HEALPix python package wrapping the *cdshealpix* Rust crate developed by FX Pineau.
- Provides healpix_to_lonlat, lonlat_to_healpix, vertices, neighbours, cone_search, polygon_search and elliptical_cone_search methods.

Cone Search	Polygon Search
	

cdshealpix: How does the binding works ?

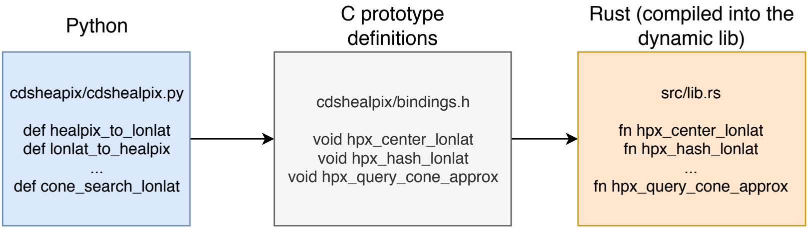


Figure 1: Python -> C -> Rust bindings

- Python sees Rust code the same way as C
- Rust functions can be **externed** as if it would be C. This is what we use for Python to call Rust functions!

cdshealpix: Python interface

- Use of *CFFI* (C Foreign Function Interface for Python) to load the dynamic library compiled (.so or .pyd for Windows) with cargo (Rust compiler)
- This is done as soon as the user imports something from **cdshealpix** (in the __*init*__.py file).

Content of cdshealpix/__init__.py

```python
import os
import sys
from cffi import FFI

ffi = FFI()
# Open and read the C function prototypes
with open(
        os.path.join(
            os.path.dirname(__file__),
            "bindings.h"
        ),
        "r") as f_in:
    ffi.cdef(f_in.read())

# Open the dynamic library generated by setuptools_rust
dyn_lib_path = find_dynamic_lib_file()
lib = ffi.dlopen(dyn_lib_path)
```

cdshealpix: Python interface

- Then **lib** and **ffi** can be imported in cdshealpix/cdshealpix.py

```python
# Beginning of cdshealpix.py
from . import lib, ffi
```

- To call Rust code, just run:

```
lib.<rust_method>(args...)
```

cdshealpix examples: lonlat_to_healpix

- Let's dive into how **lonlat_to_healpix** is wrapped around **hpx_hash_lonlat**
- **lonlat_to_healpix** in cdshealpix/cdshealpix.py

```
def lonlat_to_healpix(lon, lat, depth):
    # Handle zero dim lon, lat array cases
    lon = np.atleast_1d(lon.to_value(u.rad)).ravel()
    lat = np.atleast_1d(lat.to_value(u.rad)).ravel()

    if lon.shape != lat.shape:
        raise ValueError("The number of longitudes does \
         not match with the number of latitudes given")
```

```
num_ipixels = lon.shape[0]
# We know the size of the returned HEALPix cells
# So we allocate an array from the Python code side
ipixels = np.zeros(num_ipixels, dtype=np.uint64)
# Dynamic library call
lib.hpx_hash_lonlat(
    # depth
    depth,
    # num of ipixels
    num_ipixels,
    # lon, lat
    ffi.cast("const double*", lon.ctypes.data),
    ffi.cast("const double*", lat.ctypes.data),
    # result
    ffi.cast("uint64_t*", ipixels.ctypes.data)
)

return ipixels
```

- C hpx_hash_lonlat prototype defined in cdshealpix/bindings.h

```c
void hpx_hash_lonlat(
uint8_t depth,
uint32_t num_coords,
const double* lon,
const double* lat,
uint64_t* ipixels);
```

Rust hpx_hash_lonlat in src/lib.rs

```rust
#[no_mangle]
pub extern "C" fn hpx_hash_lonlat(
    depth: u8,
    num_coords: u32,
    lon: *const f64, lat: *const f64,
    ipixels: *mut u64,
) {
    let num_coords = num_coords as usize;

    let lon = to_slice(lon, num_coords);
    let lat = to_slice(lat, num_coords);
    let ipix = to_slice_mut(ipixels, num_coords);

    let layer = get_or_create(depth);
    for i in 0..num_coords {
        ipix[i] = layer.hash(lon[i], lat[i]);
    }
}
```

Conclusion

- Quite readable and only few lines of code:
  1. Some test exceptions
  2. One numpy array allocation
  3. The call to the dynamic library (some casts to match the C prototype)
- Whenever it is possible (size of the returned HEALPix cell array known) one should always allocate memory content in the Python side because it is auto garbage collected!
- => No need to think about free the content!
- If memory has to be allocated by the dynamic library => do not forget to call later the lib to deallocate the memory space! Let's see another example to illustrate that case !

cdshealpix examples: cone_search_lonlat

- The Python-side code does not know how much HEALPix cells will be returned by **hpx_query_cone_search**
- Thus, allocation must necessary be done in the Rust-side

Rust hpx_query_cone_search in src/lib.rs

```rust
#[no_mangle]
pub extern "C" fn hpx_query_cone_approx(
    depth: u8, delta_depth: u8,
    lon: f64, lat: f64, radius: f64
) -> *const PyBMOC {
    let bmoc = cone_coverage_approx_custom(
        depth, delta_depth, lon, lat, radius,
    );

    let cells: Vec<BMOCCell> = to_bmoc_cell_array(bmoc);
    let len = cells.len() as u32;
    // Allocation here
    let bmoc = Box::new(PyBMOC { len, cells });
    // Returns a raw pointer to a struct containing
    // * the num of HEALPix cells
    // * the array of cells
    Box::into_raw(bmoc)
}
```

- Deallocation can only be done in the Rust side too!
- Thus, Python-side must call this method

```rust
#[no_mangle]
pub extern "C" fn bmoc_free(ptr: *mut PyBMOC) {
    if !ptr.is_null() {
        unsafe {
            Box::from_raw(ptr)
            // Drop the content of the PyBMOC here.
        };
    }
}
```

- If not called, we would have memory leaks.

- This is something the Python user should not bother to do!
- Solution: wraps the result of **hpx_query_cone_approx** structure into a class

```
class ConeSearchLonLat(object):
    def __init__(self, d, delta_d, lon, lat, r):
        self.data = lib.hpx_query_cone_approx(
            d, depth_d, lon, lat, r
        )

    def __enter__(self):
        return self

    # Called when garbage collected
    def __del__(self):
        lib.bmoc_free(self.data)
        self.data = None
```

cone_search_lonlat in cdshealpix/cdshealpix.py

```python
def cone_search_lonlat(lon, lat, radius,
    depth, delta_depth):
    # Exceptions handling
    ...

    lon = lon.to_value(u.rad)
    lat = lat.to_value(u.rad)
    radius = radius.to_value(u.rad)

    cone = ConeSearchLonLat(
        depth, depth_delta,
        lon, lat, radius)
    return cone.data
```

Part III: **cdshealpix** deployment for Windows, MacOS and Linux

Setuptools_rust

- setuptools_rust package is used to:

1. Build the dynamic library (need cargo compiled installed)
2. Pack all the python files contained **cdshealpix/** + the dynamic library into a wheel

Content of the setup.py

```python
setup(...
    rust_extensions=[RustExtension(
    # Package name
    "cdshealpix.cdshealpix",
    # The path to the Cargo.toml.
    # Contains the dependencies of the Rust side code
    'Cargo.toml',
    # CFFI bindings
    binding=Binding.NoBinding,
    # --release option for cargo
    debug=False)],
    ...)
```

- `python setup.py build_wheel/install` will build the wheel into a .whl file for the host architecture (resp. install cdshealpix into your local machine)

Travis-CI

- Travis-CI is used for testing and deploying the wheels for Linux and MacOS
- The *.travis.yml* contains 2 stages: a testing & a deployment one
- Each stage is divided into jobs responsible for testing (resp. deploying) **cdshealpix** for a specific platform and python version.
- Deployment jobs use *cibuildwheel* tool. cibuildwheel uses docker with manylinux32/64bits images for generating the wheels for linux.
- See the script for deploying the wheels for linux/macos *here*.

Questions ?